



Complexité implicite : bilan et perspectives

(Implicit Computational Complexity: past and future)

mémoire présenté et soutenu publiquement le 19 octobre 2020

pour l'obtention d'une

Habilitation à Diriger des Recherches

(mention informatique, Section 27)

par

Romain Péchoux

Composition du jury

<i>Rapporteurs :</i>	Patrick Baillot	DR CNRS, LIP, ENS Lyon
	Ugo Dal Lago	PR, Università di Bologna
	Simona Ronchi Della Rocca	PR, Università di Torino
<i>Examineurs :</i>	Olivier Bournez	PR, LIX, École Polytechnique
	Emmanuel Jeandel	PR, LORIA, Université de Lorraine
	Delia Kesner	PR, IRIF, Université de Paris
	Jean-Yves Marion	PR, LORIA, Université de Lorraine

Mis en page avec la classe thesul.

Remerciements

J'aimerais remercier de nombreuses personnes, membres de ma famille, amis et collègues. J'adresse donc des remerciements les plus chaleureux et sincères:

- à ma famille, pour son soutien, et en particulier, à mon épouse, Clémence, et à mes enfants, Diane et Valère, qui me permettent, par des biais divers, de garder les pieds sur terre (surtout en période de confinement);
- à Patrick Baillot, Olivier Bournez, Ugo Dal Lago, Emmanuel Jeandel, Delia Kesner, Jean-Yves Marion et Simona Ronchi Della Rocca pour avoir accepté de siéger dans ce jury d'habilitation et, en particulier, à Patrick Baillot, Ugo Dal Lago et Simona Ronchi Della Rocca pour avoir accepté de rapporter ce manuscrit ainsi qu'à Emmanuel Jeandel pour avoir accepté d'être mon parrain scientifique ;
- aux différents collègues que j'ai pu cotoyer depuis mon recrutement dans l'équipe Inria Carte puis Mocqua, Frédéric Dupuis, Nazim Fatès, Isabelle Gnaedig, Walid Gomaa, Emmanuel Hainry, Mathieu Hoyrup, Emmanuel Jeandel, Daniel Leivant, Simon Perdrix, Donald Stull et Vladimir Zamdzhiev. Les échanges avec ses différents membres sur des sujets scientifiques et aussi politiques, philosophiques, sportifs, sociétaux et culturels ont été, sont et seront toujours un réel plaisir. Je tiens à remercier tout particulièrement Emmanuel Hainry qui m'a aidé à corriger ce manuscrit à travers une relecture minutieuse ;
- aux anciens étudiants que j'ai encadrés en thèse ou dans le cadre de stages, Hugo Férée, Pierre Mercuriali, Thanh Dinh Ta et Olivier Zeyen. Résumés en deux mots, cela donne sérieux et persévérance ;
- à tous mes collègues du laboratoire Loria et du département 2, méthodes formelles;
- aux autres étudiants croisés dans l'équipe Carte puis dans l'équipe Mocqua, Philippe Beaucamps, Titouan Carette, Alexandre Clément, Henri De Boutray, Matthieu Kaczmarek, Daniel Reynaud-Plantey, Margarita Veshchezerova et Renaud Vilmart qui ont aussi grandement contribué aux échanges de l'équipe décrits ci-dessus ;
- à l'ensemble des membres de la communauté "complexité implicite" pour tous nos échanges fructueux lors de groupes de travail, réunions diverses, workshops et conférences.

Je remercie Jean-Yves Marion, mon directeur de thèse, pour m'avoir initié à cette thématique.

Je remercie Roberto Amadio, Patrick Baillot, Neil Jones, Simona Ronchi Della Rocca, contributeurs majeurs de ce domaine, ainsi que Claude Kirchner et Paul Zimmermann, qui étaient tous membres de mon jury de doctorat.

Je remercie également les collègues du domaine avec lesquels j'ai effectué des travaux scientifiques, administratifs ou d'édition, en particulier, Martin Avanzini, Guillaume Bonfante, Marco Gaboardi, Emmanuel Hainry, Bruce Kapron, Damiano Mazza et Georg Moser.

J'ai aussi une pensée amicale et chaleureuse pour tous les anciens étudiants passionnés de cette communauté (et de son adhérence mathématique) croisés lors d'événements communautaires et devenus désormais de jeunes actifs (Beniamino Accatoli, Clément Aubert, Flavien Breuvert, Anupam Das, Laure Daviaud, Paulin De Naurois, Romain Demangeon,

Hugo Férée, Stéphane Gimenez, Alexis Ghyselen, Charles Grellois, Giulio Guerrieri, Cynthia Kop, Antoine Madet, Giulio Manzonetto, Jean-Yves Moyen, Colin Riba, Thomas Rubiano, Michael Schaper, Thomas Seiller, Florian Steinberg, Lê Thành Dũng Nguyễn, Paolo Tranquilli, Pierre Vial, Florian Zuleger, ...).

Enfin, j'ai aussi une pensée toute particulière pour la famille et les proches de Martin Hofmann, figure historique de la complexité implicite, dont la disparition tragique a endeuillé notre communauté et bien au delà ;

- à mes coauteurs scientifiques dans le domaine de la théorie des clones et de la théorie de l'agrégation, Miguel Couceiro, Erkko Lehtonen et Abdallah Saffidine.
- à l'ensemble des services (assistantes d'équipe, accueil, services informatiques, cantine, ...) de support de recherche du laboratoire Loria que j'ai côtoyés ;
- à l'ensemble des collègues de ma composante d'enseignement, l'Institut des sciences du Digital: Management & Cognition (IDMC) et aux collègues des différentes composantes d'enseignements avec lesquels j'ai collaboré (IAE Nancy, FST, Télécom Nancy, CLSH, DEG, IGA Rabat).

Je tiens particulièrement à remercier les collègues avec lesquels j'ai partagé des tâches administratives, Matthieu Barrandon, Isabelle Boni, Armelle Brun, Olivier Bruneau, Marco Dozzi, Pascal Fontaine, Olivier Perrin, Azim Roussanaly, Laurent Thomann et Laurent Vigneron, ainsi que les directeurs successifs de la composante IDMC (ex-UFR MI), Kamel Smaïli, Jeanine Souquières, Odile Thiery et, le directeur actuel, Antoine Tabbone, qui ont toujours su (ré)concilier science, exigence, recherche, enseignement et pédagogie. Je tiens à remercier tout particulièrement Jeanine et Pascal de m'avoir encouragé à écrire ce document.

Je tiens également à remercier nos services administratifs et techniques, en particulier, Virginie Besse, Marie-Luce Boulet, Marie Granddidier, Pascale Malgras et Bernard Zanga.

J'ai aussi une pensée particulière pour nos collègues disparus, Jean Malhomme et Hazel Everett.

- à tous mes autres collègues, étudiants et amis qui méritent d'être remerciés.

Contents

I Partie administrative

Formulaire de soutenance	3
Copie du diplôme de doctorat	7
Lettre de recommandation du parrain scientifique	9
Curriculum Vitae	13
1 Formation et diplômes	13
2 Parcours professionnel	13
Présentation synthétique des activités d'enseignement et d'administration	15
1 Enseignements	15
2 Responsabilités pédagogiques	16
Présentation synthétique des activités de recherche	17
1 Bilan des recherches (2004-2019)	17
2 Projets de recherche	18
3 Activités d'encadrement	20
4 Implication dans des groupes de travail et projets	21
5 Exposés invités	21
6 Responsabilités scientifiques et collectives	22
7 Logiciels	23

8	Vulgarisation	24
9	Primes et distinctions scientifiques	24
10	Collaborations nationales et internationales	24
Liste complète des publications		25
1	Articles de journaux internationaux avec comité de lecture	25
2	Articles de conférences internationales avec comité de lecture	25
3	Articles de workshops nationaux et internationaux avec comité de lecture	27
4	Doctorat	28

II Partie scientifique

Foreword	31
-----------------	-----------

Chapter 1	
Introduction	35
1.1 Computational Complexity	35
1.1.1 Description	35
1.1.2 Strengths and weaknesses	36
1.2 Implicit Computational Complexity	37
1.2.1 Description	37
1.2.2 Historical background	38
1.2.3 Function algebra and safe recursion	39
1.2.4 lambda calculus and light logics	40
1.2.5 Term rewrite systems and interpretations	43
1.2.6 Imperative programs and dataflow based methods	45
1.3 Limits	47
1.3.1 Intensionality vs decidability	47
1.3.2 Complexity of inference	48
1.4 Related Work	50
1.4.1 Termination	50
1.4.2 Computability	52

1.4.3	Finite model theory	53
1.4.4	Static analysis	53
1.5	Contribution	54

Chapter 2	
Towards a better intensionality	57

2.1	Linear logic based approaches	58
2.1.1	Light linear logic	58
2.1.2	Soft linear logic	61
2.2	Interpretations and term rewrite system	62
2.2.1	Term rewrite systems as a computational model	62
2.2.2	Interpretation methods	63
2.3	Quasi-interpretation	67
2.3.1	Motivations, definition, and basic properties	67
2.3.2	Intensional properties of quasi-interpretations	68
2.3.3	Recursive path orderings	69
2.3.4	Interpretation vs quasi-interpretation	71
2.3.5	Modularity	73
2.4	Sup-interpretation	76
2.4.1	Motivations, definition, and basic properties	76
2.4.2	Combination with the dependency pair method	78
2.4.3	Sup-interpretation vs (quasi-)interpretation	84
2.4.4	DP-interpretations for sup-interpretation synthesis	85
2.5	Summary	87

Chapter 3	
Breaking the paradigm	89

3.1	Extensions of tiering	90
3.1.1	Imperative programs	90
3.1.2	Multi-threaded programs	94
3.1.3	Fork processes	98
3.1.4	Object oriented programs	106
3.1.5	Type inference and declassification	113
3.2	Extensions of light/soft logics	114
3.2.1	Light logic and multi-threaded programs	114
3.2.2	Soft logic and process calculi	117
3.2.3	Soft linear logic and quantum programs	119

3.2.4	Miscellaneous	121
3.3	Extensions of interpretations	122
3.3.1	Higher-order rewrite systems	122
3.3.2	Functional programs	125
3.3.3	Object oriented programs	130
3.3.4	Miscellaneous	134
3.4	Extensions of other techniques	135

Chapter 4	
Extensions to infinite data types	137

4.1	Streams and quasi-interpretations	138
4.1.1	A first order lazy stream programming language	139
4.1.2	Parameterized quasi-interpretations	141
4.1.3	Stream upper bounds	141
4.1.4	Bounded input/output properties	143
4.2	Complexity class characterizations	145
4.2.1	Type-2 feasible functionals	146
4.2.2	A stream based characterization of type-2 feasible functionals	147
4.2.3	A stream based characterization of polynomial time over the reals	150
4.2.4	A higher-order characterization of feasible functionals at any type	151
4.3	Coinductive datatypes in the light affine lambda calculus	153
4.3.1	Light affine lambda calculus	154
4.3.2	Algebra and coalgebra in System F	155
4.3.3	Algebra in the light affine lambda calculus	159
4.3.4	Coalgebra in the light affine lambda calculus	160
4.4	Alternative results on streams and real numbers	163
4.4.1	Streams, parsimonious types and non-uniform complexity classes	163
4.4.2	Function algebra characterizations of polynomial time over the reals	167
4.4.3	The BSS model	168

Chapter 5	
Research perspectives	171

5.1	Probabilistic models	171
5.2	Quantum computations	172
5.3	Feasible computations over the reals	173
5.4	A decidable theory for type-2 polynomial time	173
5.5	Other typing disciplines	173

Glossary	175
List of Figures	177
Bibliography	179
Une sélection de 5 publications	201
1 Résumé des publications	201
2 Synthesis of sup-interpretations: A survey	203
3 Algebras and coalgebras in the light affine lambda calculus	246
4 Characterizing polytime complexity of stream programs using interpretations	260
5 On Bounding Space Usage of Streams Using Interpretation Analysis	286
6 A type-based complexity analysis of Object Oriented programs	334
Résumé	397
Abstract	397

I Partie administrative

Formulaire de soutenance

Demande de soutenance de l'Habilitation à Diriger des Recherches à l'Université de Lorraine

Arrêté du 23 novembre 1988 relatif à l'Habilitation à Diriger des Recherches

Formulaire à compléter et à retourner **8 semaines minimum avant la date de soutenance**

A déposer à : drv-hdr@univ-lorraine.fr

[Contact « drv-hdr » **pour information** : Direction de la Recherche et de la Valorisation (DRV), Sous-direction de l'administration de la recherche (SDAR) - 91, avenue de la Libération - BP 454 - 54001 NANCY Cedex – Mme Isabelle Maréchal - tél : 03.72.74.04.55].

I - INFORMATIONS CONCERNANT LE CANDIDAT

NOM – Prénom : M. PECHOUX ROMAIN, MCF UL, Loria, Campus Scientifique, BP 239
Vandœuvre-lès-Nancy Cedex, 06 87 38 83 59, romain.pechoux@loria.fr, section CNU 27.

Parrain scientifique : M. JEANDEL EMMANUEL, PR UL, Loria, Campus Scientifique, BP 239
Vandœuvre-lès-Nancy Cedex, 06 88 79 37 43, emmanuel.jeandel@loria.fr, section CNU 27.

II - INFORMATIONS CONCERNANT L'HDR

Ecole Doctorale IAEM

Intitulé du diplôme d'HDR : Complexité implicite : bilan et perspectives

DATE définitive de la soutenance :

19/10/2020 à 10h

Lieu : LORIA
Salle C005

Les éventuels compléments d'informations sont à communiquer le plus rapidement possible à la DRV/SDAR.

III. INFORMATIONS CONCERNANT LES RAPPORTEURS ET LE JURY

Les rapporteurs et les autres membres du jury final ci-dessous doivent être choisis au sein de la liste figurant sur le formulaire de demande d'inscription à l'HDR initialement validée par l'école doctorale. Dans le cas contraire, le présent document sera à nouveau soumis pour approbation auprès de l'ED.

Attention: le jury final devra être composé d'un minimum de 5 membres ; il est recommandé 6 ; maximum 8 (moitié de Professeurs et moitié de membres extérieurs à l'ED). En outre, l'Université de Lorraine, suite à décision du Conseil Scientifique du 19 septembre 2017, veillera à ce qu'il existe un équilibre hommes/femmes dans le jury final, en cohérence avec le secteur disciplinaire.

NOM ET QUALITÉ DES 3 RAPPORTEURS

Indiquer nom, prénom, qualité, adresse professionnelle, numéro de téléphone, e-mail, section CNU/EPST éventuelle

- M. Baillot, Patrick, Directeur de Recherche, LIP (UMR 5668), ENS Lyon, 46 Allée d'Italie, 69364, Lyon Cedex 07, France, +33 4 72 72 89 32, patrick.baillot@ens-lyon.fr, section 27, CNRS.
- M. Dal Lago, Ugo, Professor, Dipartimento di Informatica – Scienza e Ingegneria, Università degli Studi di Bologna, Mura Antea Zamboni, 7, 40127, Bologna, Italie, +39 051 2094990, ugo.dallago@unibo.it, Università degli Studi di Bologna.
- Mme Ronchi Della Rocca, Simona, Professor Emerita, Dipartimento di Informatica, Università di Torino, Corso Svizzera 185, I-10149, Torino, Italie, +39 011 6706734, ronchi@di.unito.it, Università di Torino.

NOM ET QUALITÉ DES AUTRES MEMBRES DU JURY

EXAMINATEURS :

Indiquer nom, prénom, qualité, adresse professionnelle, numéro de téléphone, e-mail, section CNU/EPST éventuelle

- M. Bournez, Olivier, Professeur, LIX (UMR 7161), Ecole Polytechnique, Laboratoire d'Informatique, 91128 Palaiseau Cedex, France, +33 1 77 57 80 78, bournez@lix.polytechnique.fr, section 27, Ecole Polytechnique.
- M. Jeandel, Emmanuel (parrain scientifique)
- Mme Kesner, Delia, Professeure, IRIF, Université Paris Diderot, case 7014, 75205 Paris Cedex 13, France, +33 1 57 27 92 38, kesner@irif.fr, section 27, Université de Paris.
- M. Marion, Jean-Yves, Professeur UL, LORIA (UMR 7503), Campus Scientifique – BP 239, 54506, Vandoeuvre-lès-Nancy, France, +33 3 83 59 20 30, jean-yves.marion@loria.fr, section 27, Université de Lorraine.

IV. VALIDATION DE LA PROPOSITION DEFINITIVE DU JURY ET DES

RAPPORTEURS PAR L'ECOLE DOCTORALE (Uniquement dans le cas où un rapporteur ou un autre membre du jury final n'est pas issu de la liste initialement validée par l'école doctorale figurant sur le formulaire de demande d'inscription à l'HDR) :

Nom, Date et signature :

Fiche synthétique à compléter obligatoirement pour diffusion de votre soutenance HDR sur le site internet de l'Université de Lorraine

Nom – Prénom	
Laboratoire de rattachement	
Intitulé du diplôme HDR	
Titre de l'HDR	

Abstract (français) – maximum 15 lignes

Dans ce mémoire, nous explorons les différents résultats obtenus par la communauté de la complexité implicite au cours de ces trente dernières années. Ces résultats permettent en général de caractériser une classe de complexité donnée sur un langage de programmation fixé en utilisant une technique d'analyse statique (interprétation, tiering, système de types, logique légère, ...). Après avoir listé les principales techniques et avoir mis en avant leurs similitudes, nous étudierons dans un premier temps, les résultats d'intensionnalité obtenus pour certaines de ces techniques, c'est-à-dire des résultats permettant de comparer le pouvoir d'expression de ces techniques ou permettant d'étendre le nombre de programmes capturés. Puis, nous étudierons des "crossovers", des extensions de ces techniques à d'autres paradigmes de programmation. Nous focaliserons ensuite notre analyse sur les résultats obtenus en utilisant ces techniques dans le cadre de domaines infinis tels que les streams, les suites infinies, les nombres réels et les fonctions d'ordre supérieur. Enfin, nous conclurons ce mémoire en exposant certaines des questions ouvertes les plus intéressantes du domaine, dont des extensions à des modèles probabilistes ou de l'informatique quantique.

Abstract (anglais) – maximum 15 lignes (pas obligatoire)

In this thesis, we explore the different results obtained by the implicit complexity community over the past thirty years. These results generally consist in characterizing a given complexity class on a fixed programming language using a static analysis technique (interpretation, tiering, type system, light logic, ...). After listing the main techniques and highlighting their similarities, we will first study the intensionality results obtained for some these techniques, i.e. results that make it possible to compare the expressive power of these techniques or to extend the number of captured programs. Then, we will study crossovers, extensions of these techniques to other programming paradigms. We will then focus our analysis on the results obtained by using these techniques in the context of infinite domains such as streams, infinite sequences, real numbers, and higher order functions. Finally, we will conclude this thesis by presenting some of the most interesting open questions in the field, including extensions to probabilistic models or quantum computing.

Copie du diplôme de doctorat

Ministère de l'enseignement supérieur et de la recherche

INSTITUT NATIONAL POLYTECHNIQUE DE LORRAINE

DOCTORAT

Vu le code de l'éducation, notamment son article L.612-7 ;

Vu le code de la recherche, notamment son article L.412-1 ;

Vu le décret n° 2002-481 du 8 avril 2002 relatif aux grades et titres universitaires et aux diplômes nationaux ;

Vu l'arrêté du 3 septembre 1998 relatif à la charte des thèses ;

Vu l'arrêté du 21 août 2000 fixant la liste des établissements d'enseignements supérieur autorisés à délivrer le doctorat conjointement avec une université ou un institut national polytechnique ;

Vu l'arrêté du 7 août 2006 relatif à la formation doctorale ;

Vu l'avis conforme du ministère de l'enseignement supérieur et de la recherche ;

Vu les pièces justificatives produites par M. ROMAIN PÉCHOUX, né le 18 décembre 1981 à BESANÇON (025), en vue de son inscription au doctorat ;

Vu le procès-verbal du jury attestant que l'intéressé a soutenu, le 14 novembre 2007 une thèse portant sur le sujet suivant : **Analyse de la complexité des programmes par interprétation sémantique**, préparée au sein de l'école doctorale Informatique-Automatique-Electronique-Electrotechnique et Mathématiques, devant un jury présidé par Simona RONCHI DELLA ROCCA, Professeur des Universités et composé de Roberto AMADIO, Professeur des Universités, Patrick BAILLOT, Chargé de Recherche, Neil JONES, Professeur des Universités, Claude KIRCHNER, Directeur de Recherche, Jean Yves MARION, Professeur des Universités, Paul ZIMMERMANN, Directeur de Recherche ;

Vu la délibération du jury ;

Le **DIPLOME DE DOCTEUR** en INFORMATIQUE

est délivré à **M. ROMAIN PÉCHOUX**

et confère le **grade de docteur**,

pour en jouir avec les droits et prérogatives qui y sont attachés.

Fait à Nancy, le 16 avril 2008

Le titulaire



Le Président

Le Recteur d'Académie,
Chancelier des universités



N° INPNAN 5809944

/2008200600116

Michel LEROY

Lettre de recommandation du parrain scientifique

Lettre de Recommandation et d'Engagement

À qui de droit,

Romain Péchoux est maître de conférences à l'Institut des Sciences du Digital : Management & Cognition (IDMC), Université de Lorraine, et au laboratoire Loria, UMR 7503, depuis 2009. Il est actuellement membre de l'équipe Mocqua dont je suis le responsable.

Les travaux de Romain portent sur la complexité implicite et, plus particulièrement, sur les techniques d'analyse statique de la complexité des programmes.

Romain a apporté de nombreuses contributions originales depuis sa thèse, dont je ne vais donner ici que deux exemples. Une des réalisations majeures de Romain concerne le développement du *tiering* à différents paradigmes de programmation. Le tiering est une méthode de typage garantissant des bornes sur le temps et l'espace nécessaires à la bonne exécution d'un programme que Romain a étendue à un langage de processus en 2013, à un langage multi-threads en 2014 et à un langage orienté objet en 2018. Ces travaux ont été implémentés en 2019 dans le logiciel COMPLEXITY-PARSER qui permet de typer des programmes Java pour garantir de telles propriétés de complexité. Sa deuxième contribution majeure correspond au développement de techniques d'analyse statique sur les langages incluant des données infinies et coinductives telles que les streams. Sur ce sujet, il a obtenu des résultats sur les langages fonctionnels à la Haskell en 2009, caractérisé le temps polynomial à l'ordre deux en 2010 et étendu le lambda calcul avec des données coinductives préservant les propriétés de normalisation en temps polynomial des logiques affines et légères en 2015 ; De par sa longue et riche expérience, Romain fait sans conteste partie des spécialistes internationaux du domaine.

Romain a clairement atteint une autonomie scientifique, comme le montre ses nombreuses collaborations nationales et internationales ; Il porte un projet de recherche, sur l'adaptation des techniques de la complexité implicite aux programmes quantiques, entièrement personnel et novateur. Il a déjà



participé à l'encadrement d'une thèse soutenue à l'Université de Lorraine, et encadre actuellement un doctorant avec Miguel Couceiro, Professeur à l'Université de Lorraine, sur des représentations minimales des fonctions Booléennes à l'aide de médianes.

Romain a également participé à l'animation scientifique de projets, a porté la responsabilité de projets internationaux (équipe associée Inria CRISTAL) et est également moteur de la communauté de la complexité implicite. Romain est actuellement coéditeur du numéro spécial de *Theoretical Computer Science* sur la complexité implicite (workshops DICE 2016, 2017 et 2018) et a participé de nombreuses fois aux comités de programme des différents workshops de cette communauté (DICE, FOPARA, RAC).

Enfin, Romain a dispensé des enseignements diversifiés dans différentes composantes de l'Université de Lorraine et, dans sa composante IDMC, il a eu la charge de responsabilités administratives importantes. En effet, Romain a eu la responsabilité de l'ensemble des enseignements du Certificat Informatique et Internet (c2i) sur l'Université de Nancy 2 de 2009 à 2011 et a exercé la fonction de directeur des études en Licence MIAGE de 2011 à 2018, démontrant ainsi qu'il sait gérer de front recherche, enseignements et responsabilités administratives.

Il ne fait donc aucun doute que Romain Péchoux a toutes les qualités attendues chez un enseignant-chercheur senior et je soutiens donc sa demande d'inscription à l'HDR, et m'engage à l'aider dans les démarches afférentes.

Nancy, 8 novembre 2019.



Emmanuel JEANDEL



Curriculum Vitae

né le 18 décembre 1981 à Besançon, marié, deux enfants,
Page web: <http://members.loria.fr/RPechoux/>
Mail : romain.pechoux@loria.fr / Téléphone : +33 (0)3 83 59 20 41

1 Formation et diplômes

2004-2007 Doctorat d'Informatique "Analyse de la complexité des programmes par interprétation sémantique", réalisé au LORIA et soutenu le 14 novembre 2007, INPL.

2001-2004 Élève ingénieur à l'école des Mines de Nancy, INPL.

2004 DÉA d'Informatique Fondamentale de l'Ecole Doctorale IAEM, mention Bien, INPL.

2004 Diplôme d'Ingénieur Civil des Mines, INPL.

2003 Maîtrise de Mathématiques, mention Assez Bien, UHP.

1999-2001 Classe Préparatoire aux Grandes Écoles, MPSI puis MP* au Lycée Victor Hugo, Besançon.

1999 Baccalauréat Scientifique, option mathématiques, mention Très Bien, Lycée du Parc, Lyon.

2 Parcours professionnel

2019-2020 Délégation CNRS dans l'Équipe Inria MOCQUA, LORIA.

2018-2019 Délégation Inria dans l'Équipe Inria MOCQUA, LORIA.

2009- Maître de conférences à l'Institut des Sciences Digitales : Management et Cognition, Université de Lorraine, et membre de l'Équipe Inria CARTE puis MOCQUA, LORIA.

2008-2009 Research fellow au Trinity College, Dublin.

2007-2008 ATER à temps complet à Telecom Nancy, UHP.

2004-2007 Moniteur à l'Université de Nancy 2 et allocataire de recherche à l'INPL - bourse MENRT - dans les Équipes Projet Inria CALLIGRAMME et CARTE au LORIA.

2004 Stage de DÉA, "Synthèse de quasi-interprétations", au LORIA, Nancy.

2003 Stage ingénieur à la direction des services informatiques de l'Office National des Forêts de Franche-Comté, Pontarlier, 2003.

2002 Stage ouvrier dans l'usine Mercedes de Daimler-Chrysler, Düsseldorf, 2002.

Présentation synthétique des activités d'enseignement et d'administration

1 Enseignements

De 2004 à 2018, j'ai occupé des fonctions de moniteur, ater et maître de conférences. J'ai dispensé les enseignements informatiques décrits ci-dessous ainsi que diverses activités d'encadrement de projets de programmation. Ces enseignements sont diversifiés tant par leur nature (théoriques et appliqués), par la diversité des niveaux auxquels ils s'adressent (de la L1 au M2), que par la pluralité des publics ciblés (lettres et droit, faculté des sciences, instituts, écoles d'ingénieurs, erasmus mundus).

2009-2018 Maître de conférences à l'Institut des Sciences Digitales : Management et Cognition (IDMC, ex-UFR MI), Université de Lorraine:

- UML, 2009-2011, formation continue, IDMC, Université de Lorraine.
- Certificat Informatique et Internet (c2i), 2009-2015, L1, Campus Lettres et Sciences Humaines et Droit-Économie-Gestion, Université de Lorraine.
- Outils logiques pour l'informatique, 2011-2018, L1 MIASHS, IDMC, Université de Lorraine.
- Programmation Java, 2015-2018, L2 MIAGE, IDMC, Université de Lorraine.
- Programmation Java, 2009-2018, L3 MIAGE, IDMC, Université de Lorraine.
- Bases de données, 2009-2018, L3 Sciences de la Gestion, ISAM-IAE, Université de Lorraine.
- Complexité algorithmique, 2011-2018, L3 MIAGE, IGA Rabat, Maroc.
- Java avancé, 2009-2015, M1 MIAGE, IDMC, Université de Lorraine.
- Mathematics for computer science (cours dispensé en langue anglaise), 2012-2018, M1 TAL et erasmus mundus, IDMC, Université de Lorraine.
- Implicit computational complexity (cours dispensé en langue anglaise), 2013-2017, M2 Informatique option LMFI et erasmus mundus, UFR STMIA, Faculté des Sciences et Technologies, Université de Lorraine.

2007-2008 ATER à temps complet, Telecom Nancy (ex-ESIAL), Université Henri Poincaré.

- Programmation Java, L1, UFR STMIA, Faculté des Sciences et Technologies, Université Henri Poincaré.
- Mathématique discrètes, 1A, Telecom Nancy, Université Henri Poincaré.

- Bases de données Oracle et programmation PL/SQL, 1A, Telecom Nancy, Université Henri Poincaré.
- Algorithmique parallèle et distribuée, 2A, Telecom Nancy, Université Henri Poincaré.
- Recherche opérationnelle, 2A, Telecom Nancy, Université Henri Poincaré.

2004-2007 Moniteur, UFR MI, Université de Nancy 2:

- Certificat Informatique et Internet (c2i), L1, Campus Lettres et Sciences Humaines et Droit-Économie-Gestion, Université de Nancy 2.
- Bases de données et programmation VB, M1 Sciences de la Gestion, ISAM-IAE, Université de Nancy 2.

2 Responsabilités pédagogiques

- Directeur du C2i (Certificat Informatique et Internet) à l'Université de Nancy 2 de 2009 à 2011. Cette responsabilité incluait la gestion administrative et pédagogique de 120 groupes d'enseignement, de 3000 étudiants et de 70 intervenants chaque semestre.
- Directeur des études en Licence MIASHS, parcours MIAGE, UFR MI, Université de Lorraine de 2011 à 2018. Cette activité incluait la gestion d'environ 90 étudiants, répartis sur deux niveaux (L2 et L3), et de 25 enseignants chaque semestre ainsi que la gestion des plannings, la mise en place des modalités de contrôles des connaissances, l'organisation des sessions d'examens, l'élaboration des maquettes, la mise en place de l'approche par compétence, de l'évaluation des enseignements et de l'auto-évaluation HCERES.
- Membre :
 - du conseil de perfectionnement de la licence MIASHS et du master MIAGE de 2011 à 2018.
 - du jury de la de la licence MIASHS et des Masters MIAGE et SCA (Sciences Cognitives et Applications), UFR MI, Université de Lorraine de 2011 à 2018.
 - du jury d'admission de la licence MIASHS et du Master MIAGE, UFR MI, Université de Lorraine de 2009 à 2018.
- Représentant à la commission d'harmonisation des recrutements d'ATER en Section 27 organisée par le LORIA en 2012.

Présentation synthétique des activités de recherche

1 Bilan des recherches (2004-2019)

Mes recherches s'inscrivent dans le domaine de la *Complexité Implicite* qui cherche à étudier la complexité de différents langages de programmation en limitant l'usage des ressources utilisées par un programme. En informatique, les ressources représentent « l'ensemble des moyens dont dispose un ordinateur pour exécuter un ou plusieurs programmes ». Les ressources informatiques sont diverses et dépendent en général du matériel et de ses limites physiques. L'informatique théorique a modélisé l'ensemble de ces contraintes physiques comme des ressources de temps et d'espace.

Ces deux ressources schématisent toutes les restrictions matérielles à l'échelle humaine. La ressource en temps va correspondre à l'exécution en temps fini d'un programme. Par exemple, un utilisateur va vouloir que son programme termine en lui retournant un résultat en un temps plus ou moins raisonnable. Cependant, l'étude de cette ressource n'est pas toujours pertinente. Par exemple, un système d'exploitation va s'exécuter en continu après que l'ordinateur a été mis sous tension et il serait dommageable qu'il s'arrête au bout d'un temps donné puisqu'il doit prendre en charge le bon fonctionnement de tous les autres programmes. De ce point de vue là, les ressources en espace sont beaucoup plus intéressantes que les ressources en temps puisque leur pertinence ne dépend pas du type de programme considéré. L'ordinateur dispose de supports de mémorisation, comme le disque dur et la mémoire vive, et leur capacité de stockage est physiquement limitée. Ainsi il sera nécessaire de contrôler la mémoire de certaines applications afin d'éviter des bogues ou des attaques. Par exemple, si l'on intègre un logiciel embarqué dans une automobile, quelle garantie possède-t-on sur le fait que le logiciel ne dépassera pas la quantité mémoire maximale disponible lors de son exécution. De telles problématiques sont primordiales dans des domaines à forts enjeux stratégiques et financiers, comme l'aérospatial et l'armement, mais elles tendent aussi à le devenir dans notre quotidien. Que faire si ma voiture ne répond plus ou si mon téléphone portable bogue après le téléchargement et l'exécution d'un jeu Java ?

Pour répondre à cette problématique, j'ai développé dans mes travaux de recherche des outils permettant d'analyser statiquement les programmes (c'est-à-dire avant exécution du programme) afin de vérifier qu'un programme vérifiant ces restrictions s'exécute en temps ou en espace donné (propriété de correction). Réciproquement, j'ai aussi vérifié que toute fonction qui se calcule en temps ou en espace donné peut être calculée par un programme vérifiant ces restrictions (propriété de complétude). Ces résultats sont principalement de nature théorique mais peuvent être appliqués à des langages de programmation usuels (CAML, HASKELL, LISP, JAVA, SCALA, ...). Ils peuvent se diviser entre les différents paradigmes de programmation pour lesquels j'ai fourni une telle analyse :

- Systèmes de réécriture : Publications [1,2,4,9,10,11,12,13,17,28,29]
- Programmes fonctionnels : Publications [3,7,15,16,22,23,27,34,35]
- Programmes impératifs/orientés-objet : Publications [5,14,21,26,30,31,33,38,41]
- Programmes parallèles/concurrents : Publications [18,19,32]

Par ailleurs certaines de mes publications traitent de l’extension de ces outils à des programmes utilisant des streams (des flux infinis de données) qui sont, par exemple, utilisés pour traiter l’échange de données sur Internet (Publications : [3,4,15,16,17,22,27]).

J’ai aussi obtenu une publication dans le domaine de la virologie informatique. Ce papier traite de techniques d’obfuscation afin de détecter des attaques virales sur les programmes (Publication : [20]). Et j’ai obtenu différentes publications sur la complexité de la représentation de fonctions booléennes dans le cadre de la théorie des clones (Publications : [6,8,24,36,37]). Enfin, je commence à obtenir des résultats dans le domaine de l’informatique quantique (Publications : [25,39,40]) dans le cadre de mes années de délégation Inria et de délégation CNRS.

2 Projets de recherche

2.1 Analyse de la complexité des programmes quantiques

Motivations.

Ce premier projet s’insère directement dans la continuité de la création de la nouvelle Équipe MOCQUA, dont l’objectif est d’étudier les problématiques liées aux nouveaux modèles de calcul, et de l’acceptation du Projet anr SoftQPro, débuté en 2018 en collaboration avec Atos (Bull), le CEA et l’Université d’Orsay, dont l’objectif est d’étudier les programmes quantiques et d’optimiser leurs ressources.

Le fait que les ordinateurs quantiques permettent de calculer certains algorithmes en un temps plus court que les machines classiques est un phénomène connu depuis de nombreuses années. Pour de nombreux algorithmes classiques, un algorithme quantique sémantiquement équivalent a été développé (Shor, Grover, ...) et il a été démontré que sa complexité est meilleure.

L’identification des ressources qui permettent à l’ordinateur quantique d’être plus efficace que sa contrepartie classique reste cependant un problème ouvert dont la résolution dépend directement du modèle quantique étudié. En particulier, le rôle de l’intrication (propriété d’interdépendance entre deux états quantiques) dans ce processus n’a pas encore clairement été établi. De plus, le récent développement de prototypes quantiques (Quipper, QuaML, ...) rend désormais possible l’analyse pratique des ressources.

Contributions.

Mon projet consiste à développer des techniques pour certifier l’utilisation des ressources du langage quantique développé dans le cadre de l’anr SoftQPro: QuaML. Ces techniques seront basées sur l’analyse statique, des transformations de programmes ainsi que des méthodes de certification et d’optimisation de programme. Il s’agit donc de permettre une interaction entre différents domaines de l’informatique: l’algorithmique quantique, les langages de programmation et les méthodes formelles.

Parmi les différentes techniques que j’envisage pour contrôler les ressources des programmes quantiques, les deux directions les plus prometteuses sont basées sur l’utilisation d’interprétations abstraites et sur l’étude de la complexité implicite des programmes.

- Les interprétations abstraites ont été développées par Patrick Cousot et ses coauteurs pour approximer la sémantique des programmes. Alors que la sémantique usuelle d'un programme permet de décrire son comportement, une sémantique à base d'interprétation abstraite permet de la simplifier en interprétant le comportement dans un domaine plus restreint et plus simple à analyser. Les interprétations abstraites ont été utilisées pour l'analyse statique, le débogage ainsi que la certification de programmes. Je propose de les utiliser pour étudier les niveaux d'intrication de la mémoire quantique durant l'exécution d'un programme et de poursuivre les travaux que nous venons de débiter avec Simon Perdrix, CR CNRS, et Vladimir Zamdzhiev, post-doctorant, sur la sémantique des langages de programmation quantiques avec types inductifs.
- La complexité implicite étudie la complexité des programmes indépendamment du modèle de calcul, soit en restreignant la syntaxe du langage (opérateurs de base, structures et schémas de récurrence), soit en appliquant un critère d'analyse statique tel que des systèmes de type, des contraintes ou des interprétations. Dans ce dernier cas de figure, si un programme vérifie un critère fixé (par exemple, le fait d'être typable pour un système de type donné) alors on peut garantir des bornes supérieures sur l'utilisation de ses ressources en temps et en espace. L'étude de programmes quantiques avec ces techniques est un problème complètement ouvert : seules 3 publications par Ugo Dal Lago, Université de Bologne, et ses coauteurs existent à ce jour (avec mémoire quantique mais sans flot de contrôle quantique). Je propose d'adapter ces techniques à l'analyse en temps et espace de vrais programmes quantiques. Cette adaptation est loin d'être une formalité à partir du moment où le programme dispose d'un flot de contrôle quantique (on est alors plus proche de modèles probabilistes puisque le calcul du programme peut être une superposition de différents états quantiques).

2.2 Complexité des langages de streams.

Dans le cadre des ANR Complice puis Elica, j'ai commencé à travailler avec Emmanuel Hainry et Marco Gaboardi, associate professor, Boston University, US, sur les langages de streams. Nous cherchons à adapter des systèmes de types basés sur la logique linéaire à des langages fonctionnels sur les streams (suites infinies), à la Haskell, afin de contrôler la complexité des calculs.

Un programme sur les streams est productif s'il peut être évalué de manière continue en un stream infini, la limite du calcul. Cette propriété est largement étudiée depuis une trentaine d'années. Nous cherchons à aller plus loin en garantissant une propriété plus forte: le programme doit être productif mais aussi générer son n -ème élément de sortie en temps polynomial en n , pour tout n . A cet effet, nous cherchons à développer des systèmes de type garantissant une telle propriété.

Les applications d'un tel système de type sont très intéressantes. En effet, sur un stream binaire signé (vu comme un nombre réel), il permettrait de capturer les programmes calculant sur les réels en temps polynomial (au sens de Ko). Nous pourrions ainsi définir des bibliothèques de programmes fonctionnels efficaces qui calculent sur les réels. Par ailleurs, ces systèmes doivent pouvoir être adaptés pour capturer l'espace logarithmique sur les streams. Cette caractérisation correspond aux programmes fonctionnant en temps infini mais avec un espace de travail raisonnable et est très proche en pratique des applications fonctionnant sur un smartphone (l'application ne termine pas mais est contrainte par un espace mémoire très restreint).

2.3 Equivalence contextuelle pour la complexité.

J'ai débuté une collaboration avec Vassilis Koutavas, Associate Professor au Trinity College, Dublin. Notre objectif est d'adapter les travaux existants concernant l'équivalence contextuelle de programmes afin d'étudier leur complexité. Deux programmes sont contextuellement équivalents si pour tout contexte, leur terminaison est équivalente. L'équivalence contextuelle peut être définie en terme de bisimulation. Nous cherchons à modifier cette notion pour qu'elle ne concerne plus seulement la terminaison mais la complexité.

Un tel outil nous permettrait d'augmenter énormément l'expressivité de certains outils de la complexité implicite. En effet, certaines techniques sont limitées dans la mesure où elles ne sont pas closes par composition. L'équivalence contextuelle viendrait réparer ce défaut : Par exemple, si un programme p est équivalent à un programme p' qui termine en temps quadratique alors nous pourrions inférer que p termine en temps quadratique.

2.4 Représentations optimales des fonctions booléennes.

Je coencadre actuellement avec Miguel Couceiro, PR UL, un doctorant, Pierre Mercuriali, qui travaille sur la théorie de l'agrégation. Cette théorie s'intéresse à l'extraction d'informations en général. Par exemple, comment extraire des informations pertinentes à partir de réponses de différentes personnes à un questionnaire en ligne ? À cet effet, nous étudions des représentations optimales en taille des fonctions booléennes (termes, circuits, ...) et nous les comparons pour trouver les systèmes les plus efficaces. Par exemple, nous avons démontré que des termes utilisant la médiane sont strictement plus efficaces que des termes en CNF et DNF.

J'envisage de poursuivre ces travaux ainsi que les collaborations déjà établies avec des collègues de l'Université de Dresden, de l'EPFL, Lausanne et de l'Université de Sidney.

3 Activités d'encadrement

- Post-doctorat :
 - Vladimir Zamdzhiev, depuis octobre 2018, avec Simon Perdrix, CR CNRS (50%).
Publications communes : [25,39,40].
- Doctorat :
 - Pierre Mercuriali, depuis 2016, contrat doctoral, avec Miguel Couceiro, PR UL (50%).
Publications communes : [6,8,24,36].
 - Thanh Dinh Ta, de 2011 à 2013, bourse Inria cordiS, avec Jean-Yves Marion, PR UL (50%).
Publications communes : [19].
Thanh Dinh est désormais ingénieur de recherche chez Tetrane (www.tetrane.com).
- Master :
 - Pierre Mercuriali, Master 2, Université de Lorraine, 2016, avec Miguel Couceiro, PR UL.
Publications communes : [6,8,24,36].
 - Gwendal Carpy, Master 2, École des Mines de Nancy, 2014, avec Émmanuel Hainry, MCF UL.
Gwendal est désormais ingénieur chez Elca (www.elca.ch).

- Hugo Férée, Master 1 et Master 2, ENS Lyon, 2010 et 2011, avec Émmanuel Hainry, MCF UL, et Mathieu Hoyrup, CR Inria.
Publications communes : [4,17].
Hugo est désormais maître de conférences à l’Université Paris Diderot.
- Thanh Dinh Ta, Master 2, IFI Hanoi et Université Claude Bernard, 2010, avec Jean-Yves Marion, PR UL.
Publications communes : [19].
Thanh Dinh est désormais ingénieur de recherche chez Tetrane (www.tetrane.com).
- Licence :
 - Olivier Zeyen, Licence 3, FST UL, stage de 3 mois, 2019.

4 Implication dans des groupes de travail et projets

- Membre du :
 - Projet CRISS de l’ACI “SÉCURITÉ INFORMATIQUE”, 2003-2006.
 - GT CALCULABILITÉ du GDR IM.
 - GT SCALP du GDR IM.
 - GT IQ du GDR IM.
 - Projet COMPLICE de l’ANR BLANC ANR-08-BLAN-0211, 2008-2014.
 - Projet BINSEC de l’ANR-12-INSE-0002, 2012-2014.
 - Projet ELICA de l’ANR-14-CE25-0005, 2014-2019.
 - Projet SOFTQPRO de l’ANR PRCE, 2018-2022.
- Investigateur principal :
 - de l’équipe associée Inria, CRISTAL (Contrôle des Ressources par Interprétation Sémantique et Typages Affine et Linéaire) en collaboration avec l’équipe CARTE et le département méthodes formelles de l’université de Turin, Italie, de 2009 à 2012.
 - de l’équipe associée Inria, TC(Pro)³ (Propriétés de Terminaison et de Complexité des Programmes Probabilistes) en collaboration avec l’équipe MOCQUA et l’Université d’Innsbruck, Autriche, depuis 2020.

5 Exposés invités

J’ai donné les exposés invités suivant:

- Sup-interpretations, Follia Project, Workshop on Implicit Computational Complexity, 18 janvier 2006, Turin, Italie.
- Program complexity analysis by interpretation, séminaire de l’équipe LANDE, IRISA, 7 février 2008, Rennes, France.
- Program complexity analysis by interpretation, séminaire de l’équipe POPS, Inria Lille-Nord Europe, 3 juin 2008, Lille, France.

- Program complexity analysis by interpretation, séminaire du LIFO, LIFO, 30 juin 2008, Orléans, France.
- Semantics interpretation, Foundations@Lero, Trinity College, 17 octobre 2008, Dublin, Irlande.
- Interpretation of stream programs, Workshop on Computer Science, E-JUST University, 24 octobre 2010, Alexandrie, Égypte.
- A type system for complexity flow analysis of imperative programs, PLClub, University of Pennsylvania, 24 février 2012, Philadelphie, États-Unis.
- A type system for analyzing the complexity of Object Oriented programs, research Seminar, University of Dundee, 6 mars 2014, Dundee, Écosse.
- Tiering and non-interference for complexity analysis of imperative and OO programs, Mathematical Structures of Computation workshop et séminaire Chocola, 12 février 2014, ENS Lyon, France.
- Algebra and Coalgebra in the Light Affine Lambda Calculus, Mathematics for Decision and Discovery (M4D2), séminaire Malotec, 11 may 2016, Loria, France.
- A tier-based typed programming language characterizing feasible functionals, Shonan seminar #151: Higher-order Complexity Theory and its Applications, 7 octobre 2019, Shonan Village, Japan.

6 Responsabilités scientifiques et collectives

- Éditeur invité pour Theoretical Computer Science (Special Issue on Implicit Computational Complexity: open post-conference publication of the workshops DICE 2016, 2017 and 2018).
- Évaluateur expert pour :
 - l'appel à projets INdAM-cofund-2012 cofinancé par l'Instituto Nazionale di Alta Matematica et par Marie Curie FP7 Program,
 - l'appel à projets H2020 Marie Skłodowska-Curie Actions, Individual Fellowships, 2018.
- Rapporteur et évaluateur expert pour l'appel à projets H2020 Marie Skłodowska-Curie Actions, Individual Fellowships, 2019.
- Membre du comité d'organisation :
 - des workshops TCV 2006 et 2007,
 - de la conférence Computability and Complexity in Analysis, CCA 2013.
- Membre du comité de programme des workshops :
 - DICE 2014, DICE 2018 et DICE 2019,
 - FOPARA 2015 et FOPARA 2019,

- RAC 2016.
- Relecteur pour les journaux internationaux :
 - ACM Transactions on Computational Logic,
 - AMS Mathematical Review,
 - Computability,
 - Information & Computation,
 - Information Processing Letters.
 - Journal of Automated Reasoning,
 - Theoretical Computer Science.
- Relecteur pour les conférences et workshops internationaux :
 - CONCUR 2008,
 - CSL 2012, 2018
 - DICE 2014, 2018, 2019,
 - FOPARA 2015, 2019,
 - FOSSACS 2006, 2016, 2019,
 - ICALP 2019, 2020.
 - ISMVL 2015-2020,
 - LICS 2020,
 - LFA 2016,
 - MFCS 2012,
 - RAC 2016,
 - RTA 2008-2010, 2014,
 - STACS 2008, 2017
 - TLCA 2014,
 - WORDS 2013,
- Membre élu du conseil du laboratoire LORIA depuis février 2018.
- Membre élu du conseil de l’Institut des Sciences Digitales : Management et Cognition de 2014 à 2018.
- Responsable de l’équipe associée Inria, CRISTAL de 2009 à 2012.
- Responsable du rapport d’activités 2015 de l’équipe projet INRIA Carte.

7 Logiciels

COMPLEXITYPARSER. Analyseur statique de la complexité des programmes Java basé sur la technique du tiering. Logiciel écrit en Java (environ 5000 lignes) et réalisé en collaboration avec Emmanuel Jeandel, Emmanuel Hainry et Olivier Zeyen, 2019.

8 Vulgarisation

Depuis 2015, je suis relecteur pour Mathematical Reviews, American Mathematical Society. L'objectif de ces relectures est de fournir des critiques, des synthèses et des résumés scientifiques d'articles destinés à aider le lecteur dans sa compréhension. J'ai à ce titre déjà rédigé 3 reviews.

9 Primes et distinctions scientifiques

J'ai obtenu la Prime d'Encadrement Doctoral et de Recherche en 2017.

10 Collaborations nationales et internationales

Cette section décrit la liste de mes collaborateurs, classée par domaine, avec lesquels j'ai des travaux en cours :

- Equivalence contextuelle :
 - Vassilis Koutavas, associate professor, Trinity College, Irlande.

- Complexité Implicite :
 - Martin Avanzini, CR Inria, Sophia Antipolis,
 - Marco Gaboardi, associate professor, Boston University, US,
 - Emmanuel Hainry, MCF UL, LORIA,
 - Jean-Yves Marion, PR UL et directeur du LORIA,
 - Damiano Mazza, CR HDR CNRS, LIPN,
 - Georg Moser, PR, University of Innsbruck, Autriche,
 - Bruce Kapron, PR, Victoria University, Canada.

- Théorie des clones et circuits Booléens :
 - Miguel Couceiro, PR UL, LORIA.
 - Erkko Lehtonen, PostDoc, TU Dresden, Allemagne,
 - Abdallah Saffidine, associate professor, University of Sydney, Australie,
 - Mathias Soeken, CR, EPFL, Suisse.

- Informatique quantique :
 - Simon Perdrix, CR HDR CNRS, LORIA,
 - Mathys Rennela, PostDoc, Leiden University, Pays-Bas,
 - Vladimir Zamdzhiev, PostDoc, LORIA.

Liste complète des publications

1 Articles de journaux internationaux avec comité de lecture

1. Jean-Yves Marion and Romain Péchoux. Sup-interpretations, a semantic method for static analysis of program resources. *ACM Transactions on Computational Logic*, 10(4), 31 pages, 2009.
2. Romain Péchoux. Synthesis of sup-interpretations: A survey. *Theoretical Computer Science*, 467:30–52, 2013.
3. Marco Gaboardi and Romain Péchoux. On Bounding Space Usage of Streams Using Interpretation Analysis. *Science of Computer Programming*, 111(3):395–425, 2015.
4. Hugo Férée, Emmanuel Hainry, Mathieu Hoyrup, and Romain Péchoux. Characterizing polynomial time complexity of stream programs using interpretations. *Theoretical Computer Science*, 585:41–54, 2015.
5. Emmanuel Hainry and Romain Péchoux. A Type-Based Complexity Analysis of Object Oriented Programs. *Information and Computation*, Information and Computation, 261:78–115, 2018.
6. Miguel Couceiro, Pierre Mercuriali, Romain Péchoux and Abdallah Saffidine. On the complexity of minimizing median normal forms of monotone Boolean functions and lattice polynomials. *Journal of Multiple Valued Logic and Soft Computing*, 33(3):197-218, 2019.
7. Emmanuel Hainry and Romain Péchoux. Theory of higher-order interpretations and application to Basic Feasible Functions. Conditionally accepted under minor revision to *Logical Methods in Computer Science*.
8. Miguel Couceiro, Erkko, Lehtonen, Pierre Mercuriali, and Romain Péchoux. On the efficiency of normal form systems for representing Boolean functions. *Theoretical Computer Science*, 813: 341–361, 2020.

2 Articles de conférences internationales avec comité de lecture

9. Guillaume Bonfante and Jean-Yves Marion and Romain Péchoux. A Characterization of Alternating Log Time by First Order Functional Programs. In *Logic for Programming, Artificial Intelligence, and Reasoning, 13th International Conference, LPAR 2006, Phnom Penh, Cambodia, November 13-17, 2006, Proceedings*, pages 90–104, 2006. Taux d'acceptation : 39,53%.

10. Jean-Yves Marion and Romain Péchoux. Resource Analysis by Sup-interpretation. In *Functional and Logic Programming, 8th International Symposium, FLOPS 2006, Fuji-Susono, Japan, April 24-26, 2006, Proceedings*, pages 163–176, 2006. Taux d’acceptation : 33,33%.
11. Guillaume Bonfante and Jean-Yves Marion and Romain Péchoux. Quasi-interpretation Synthesis by Decomposition. In *Theoretical Aspects of Computing - ICTAC 2007, 4th International Colloquium, Macau, China, September 26-28, 2007, Proceedings*, pages 410–424, 2007. Taux d’acceptation : 42,02%.
12. Jean-Yves Marion and Romain Péchoux. A Characterization of NCK. In *Theory and Applications of Models of Computation, 5th International Conference, TAMC 2008, Xi’an, China, April 25-29, 2008. Proceedings*, pages 136–147, 2008. Taux d’acceptation : 26,04%.
13. Jean-Yves Marion and Romain Péchoux. Characterizations of polynomial complexity classes with a better intensionality. In *Proceedings of the 10th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 15-17, 2008, Valencia, Spain*, pages 79–88, 2008. Taux d’acceptation : 47,05%.
14. Jean-Yves Marion and Romain Péchoux. Analyzing the Implicit Computational Complexity of object-oriented programs. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2008, December 9-11, 2008, Bangalore, India*, pages 316–327, 2008. Taux d’acceptation : 29,05%.
15. Marco Gaboardi and Romain Péchoux. Global and Local Space Properties of Stream Programs. In *Foundational and Practical Aspects of Resource Analysis - First International Workshop, FOPARA 2009, Eindhoven, The Netherlands, November 6, 2009, Revised Selected Papers*, pages 51–66, 2009.
16. Marco Gaboardi and Romain Péchoux. Upper Bounds on Stream I/O Using Semantic Interpretations. In *Computer Science Logic, 23rd international Conference, CSL 2009, 18th Annual Conference of the EACSL, Coimbra, Portugal, September 7-11, 2009. Proceedings*, pages 271–286, 2009. Taux d’acceptation : 38,02%.
17. Hugo Férée, Emmanuel Hainry, Mathieu Hoyrup, Romain Péchoux. Interpretation of Stream Programs: Characterizing Type 2 Polynomial Time Complexity. In *Algorithms and Computation - 21st International Symposium, ISAAC 2010, Jeju Island, Korea, December 15-17, 2010, Proceedings, Part I, Proceedings*, pages 291–303, Lecture Notes in Computer Science, Springer, 2010. Taux d’acceptation : 42,30%.
18. Emmanuel Hainry, Jean-Yves Marion, and Romain Péchoux. Type-based complexity analysis for fork processes. In *the 16th International Conference on Foundations of Software Science and Computation Structures, FOSSACS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, pages 305–320, Lecture Notes in Computer Science, Springer, 2013. Taux d’acceptation : 25,68%.
19. Jean-Yves Marion and Romain Péchoux. Complexity information flow in a multi-threaded imperative language. In *the 11th Annual Conference on Theory and Applications of Models of Computation, TAMC 2014, Chennai, India, April 11-13, 2014. Proceedings*, pages 124–140, Lecture Notes in Computer Science, Springer, 2014. Taux d’acceptation : 24,10%.

20. Romain Péchoux and Thanh Dinh Ta. A categorical treatment of malicious behavioral obfuscation. In *the 11th Annual Conference on Theory and Applications of Models of Computation, TAMC 2014, Chennai, India, April 11-13, 2014. Proceedings*, pages 280–299, Lecture Notes in Computer Science, Springer, 2014. Taux d’acceptation : 24,10%.
21. Emmanuel Hainry and Romain Péchoux. Objects in Polynomial Time. In *the 13th Asian Symposium on Programming Languages and Systems, APLAS 2015, Pohang, South Korea, November 30- December 2, 2015. Proceedings*, pages 387–404, Lecture Notes in Computer Science, Springer, 2015. Taux d’acceptation : 32,43%.
22. Marco Gaboardi and Romain Péchoux. Algebras and coalgebras in the light affine lambda calculus. In *the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015. Proceedings*, pages 114–126, ACM, 2015. Taux d’acceptation : 29,41%.
23. Emmanuel Hainry and Romain Péchoux. Higher-order interpretations for higher-order complexity. In *the 21st International Conference on Logic for Programming Artificial Intelligence and Reasoning, LPAR 2017, Maun, Botswana*. Taux d’acceptation : 41,02%.
24. Miguel Couceiro, Pierre Mercuriali, Romain Péchoux and Abdallah Saffidine. Median based calculus for lattice polynomials and monotone Boolean functions. In *the IEEE 47th International Symposium on Multiple-Valued Logic, ISMVL 2017, Novi Sad, Serbia*. Taux d’acceptation : 73,07%.
25. Romain Péchoux, Simon Perdrix, Mathys Rennela, and Vladimir Zamdzhiev. Quantum Programming with Inductive Datatypes: Causality and Affine Type Theory. In *Foundations of Software Science and Computation Structures - 23rd International Conference, FOSSACS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*, pages 562–581, Lecture Notes in Computer Science, Springer, 2020.
26. Emmanuel Hainry, Bruce Kapron, Jean-Yves Marion, and Romain Péchoux. A tier-based typed programming language characterizing Feasible Functionals. In *LICS '20: 35th Annual ACM/IEEE Symposium on Logic in Computer Science, Saarbrücken, Germany, July 8-11, 2020*, pages 535–549, ACM, 2020.
27. Emmanuel Hainry, Damiano Mazza, and Romain Péchoux. Polynomial time over the reals with parsimony. In *15th International Symposium on Functional and Logic Programming, FLOPS 2020, Akita, Japan, September 14-16, 2020*, ACM, 2020.

3 **Articles de workshops nationaux et internationaux avec comité de lecture**

28. Guillaume Bonfante, Jean-Yves Moyen, Jean-Yves Marion and Romain Péchoux. Quasi-interpretation synthesis. In *Logic and Complexity in Computer Science, LCC 2005*, Chicago, US, June 2005.
29. Jean-Yves Marion and Romain Péchoux. Quasi-friendly sup-interpretations. In *Logic and Complexity in Computer Science, LCC 2006*, Wroclaw, Poland, July 2006.
30. Jean-Yves Marion and Romain Péchoux. Resource control of object-oriented programs. In *Logic and Complexity in Computer Science, LCC 2007*, Seattle, US, August 2007.

31. Emmanuel Hainry and Romain Péchoux. Types for controlling heap and stack in Java. In *Third International Workshop on Foundational and Practical Aspects of Resource Analysis, FOPARA 2013*, Bertinoro, Italy, August 2013.
32. Romain Péchoux. Bounding Reactions in the Pi-calculus using Interpretations. In *Third International Workshop on Foundational and Practical Aspects of Resource Analysis, FOPARA 2013*, Bertinoro, Italy, August 2013.
33. Emmanuel Hainry and Romain Péchoux. Implicit computational complexity in Object Oriented Programs. In *Developments in Implicit Computational Complexity, DICE 2015*, London, United Kingdom, April 2015.
34. Emmanuel Hainry and Romain Péchoux. Higher-order interpretations for Basic Feasible Functions. In *Developments in Implicit Computational Complexity, DICE 2015*, London, United Kingdom, April 2015.
35. Emmanuel Hainry and Romain Péchoux. Higher-order interpretations for higher-order complexity. In *Developments in Implicit Computational Complexity, DICE-FOPARA 2017*, Uppsala, Sweden, April 2017.
36. Miguel Couceiro, Pierre Mercuriali and Romain Péchoux. On the efficiency of normal form systems of Boolean functions. In *Rencontres Francophones sur la Logique Floue et ses Applications, LFA 2017*, Amiens, France, October 2017.
37. Miguel Couceiro, Erkki Lehtonen, Pierre Mercuriali, Romain Péchoux and Mathias Soeken. Normal form systems generated by single connectives have mutually equivalent efficiency. In *Developments in Implicit Computational Complexity, DICE 2018*, Thessaloniki, Greece, April 2018.
38. Emmanuel Hainry, Bruce Kapron, Jean-Yves Marion, and Romain Péchoux. Tiered complexity at higher-order. In *Developments in Implicit Computational Complexity, DICE 2019*, Praha, Czech Republic, April, 2019.
39. Romain Péchoux, Simon Perdrix, Mathys Rennela and Vladimir Zamdzhiev. Quantum Programming with Inductive Datatypes: Causality and Affine Type Theory (Early Idea). In *8th Conference on Algebra and Coalgebra in Computer Science, CALCO 2019*, London, United Kingdom, June, 2019.
40. Romain Péchoux, Simon Perdrix, Mathys Rennela and Vladimir Zamdzhiev. Inductive Datatypes for Quantum Programming. In *Applied Category Theory Conference, ACT 2019*, Oxford, United Kingdom, July, 2019.
41. Emmanuel Hainry, Bruce Kapron, Jean-Yves Marion, and Romain Péchoux. Tiered complexity at higher-order. In *Third Workshop on Mathematical Logic and its Applications, MLA 2019*, Nancy, France, January, 2019.

4 Doctorat

42. Romain Péchoux. Analyse de la complexité des programmes par interprétation sémantique. (Program complexity analysis by semantics interpretation). Institut National Polytechnique de Lorraine, Nancy, France, 187 pages, 2007.

II Partie scientifique

Foreword

This part of the document attempts to survey the distinct Implicit Computational Complexity results obtained in the last three decades, focusing not only on the author's results and trying to be as exhaustive as possible by presenting the main advances obtained in the field. For this to be presented in a concise way, some arbitrary choices had been performed and some lines of work could have been put aside or not described in full details with all results.

This document is not introductory to Implicit Computational Complexity (ICC) and it is assumed that the reader is familiar with the following fields of theoretical computer science:

- *Computability theory*. General notions of decidability/undecidability, computability/un-computability and computational models such as Turing Machines (TM) or Random Access Machines (RAM) will be used throughout the document. An introduction to these notions can be found in [Sav98].
- *Computational complexity*. This document discusses various characterizations of well-known complexity classes such as `Alogtime`, `NC`, `(F)P`, `NP`, `(F)PSPACE` and `EXPTIME`. [AB09] provides an interesting overview of the subject.

Some more technical and specific knowledge on the following programming languages/computational models and their semantics is required.

- *Function algebra* are used in Chapter 1, Chapter 3, and Chapter 4. Their definition and knowledge on existing characterizations based on such formalism will help the reader.
- *Lambda calculus* is used throughout the document. Basic knowledge about its syntax and semantics is required. Knowledge about the simply typed lambda calculus and System F will also be helpful.
- *Imperative programs* are used in Chapter 1 and Chapter 3. The language under study is an abstract and simple language. Some knowledge on processes and multi-threads will also be required in Chapter 3.
- *Term rewrite systems* are used throughout the document. We provide some very brief introduction to this formalism at the beginning of Section 2.2.

Some extra and specific knowledge about *object oriented programs* and specialized computational models such as the π -calculus, *quantum programs*, the *Blum-Shub and Smale model*, and *computable analysis* will be required in Chapter 3 and Chapter 4. For OO programs, basic knowledge in Java programming and about the formal semantics of such languages, *e.g.* FeatherweightJava, is enough to get a full understanding.

For each of these formalisms, we provide the corresponding sections and a complementary reading of interest in Table 1.

Programming language	1.2	1.3	1.4	2.1	2.2	2.3	2.4	3.1	3.2	3.3	4.1	4.2	4.3	4.4	Reference
Function algebra	✓	✓						✓						✓	[Clo99]
Lambda calculus	✓	✓	✓	✓	✓				✓	✓	✓	✓	✓	✓	[B+84]
Imperative programs	✓	✓	✓	✓				✓	✓						[Win93]
Term Rewrite Systems	✓	✓	✓	✓		✓	✓			✓		✓			[BN99, K+01]
Object Oriented programs								✓		✓					[Pie02]
π -calculus									✓						[SW03]
Quantum programs									✓						[Sel04]
Computable analysis											✓			✓	[Wei00]
Blum-Shub and Smale model														✓	[BSS88]

Table 1: Table of prerequisites on programming languages and computational models per section

Technique	1.2	1.3	1.4	2.1	2.2	2.3	2.4	3.1	3.2	3.3	4.1	4.2	4.3	4.4	Reference
Interpretations	✓	✓	✓	✓	✓	✓	✓			✓	✓	✓			[BMM11, Péc13]
Type systems	✓	✓	✓	✓	✓			✓	✓				✓	✓	[Pie02]
Linear logic	✓	✓		✓					✓				✓	✓	[Gir87]
Termination techniques	✓		✓		✓	✓	✓								[BN99, K+01, AG00]
Category theory									✓				✓		[Pie91]
Algebra and coalgebra													✓		[JR97]
Higher-order complexity												✓			[JRK01]
Complexity of real functions											✓		✓		[K91]

Table 2: Table of prerequisite on tools and techniques per section

Some specific knowledge on the following tools/techniques is also required.

- *Interpretations, type systems, and linear logic* are the main tools used throughout the document.
- *Termination* techniques are used in Chapter 1 and Chapter 2. Knowledge on Recursive Path Orders (RPO) and Dependency Pairs (DP) will help the reader in understanding the corresponding sections.
- Some basic on *category theory* will be used in Chapter 3 and Chapter 4. Categorical knowledge on the notions of *algebra and coalgebra* will also be helpful in Chapter 4.
- Some knowledge on higher-order complexity will be helpful in Chapter 4. Some basic notions are provided in Section 4.2.
- Some knowledge on the complexity of real functions, *i.e.* functions over \mathbb{R} , will also be required in Chapter 4.

For each of these tools/techniques, we provide the corresponding sections and a complementary reading of interest in Table 2.

Chapter 1

Introduction

Contents

1.1 Computational Complexity	35
1.1.1 Description	35
1.1.2 Strengths and weaknesses	36
1.2 Implicit Computational Complexity	37
1.2.1 Description	37
1.2.2 Historical background	38
1.2.3 Function algebra and safe recursion	39
1.2.4 lambda calculus and light logics	40
1.2.5 Term rewrite systems and interpretations	43
1.2.6 Imperative programs and dataflow based methods	45
1.3 Limits	47
1.3.1 Intensionality vs decidability	47
1.3.2 Complexity of inference	48
1.4 Related Work	50
1.4.1 Termination	50
1.4.2 Computability	52
1.4.3 Finite model theory	53
1.4.4 Static analysis	53
1.5 Contribution	54

1.1 Computational Complexity

1.1.1 Description

Computational complexity is the discipline of classifying functions depending on their inherent difficulty or cost. In this field, machines, more specifically Turing Machines (TM), are used as standard computational models for estimating the degree of difficulty of a function. Machines are compared by relating the cost (sometimes referred as Blum complexity measure [Blu67]) needed to produce an output *i.e.*, to reach a final state, to the input size (the number of non empty cells on the input tape at the beginning of the execution), for any possible input. Examples of such costs are the time, defined as the number of transitions required for the execution to complete,

or the space, defined as the maximal number of non empty memory cells at any time during the execution.¹

The comparison between machines can be extended to functions. Given a function $t : \mathbb{N} \rightarrow \mathbb{N}$, a function f is computable with cost $t(n)$ if there exists a machine computing f with cost at most $t(n)$ for any input of size n .

Focusing on time cost, FP is defined to be the class of functions computable in polynomial time by a deterministic TM. A function f is in FP if there exist a deterministic machine M and a polynomial P such that M computes f at cost at most $P(n)$, for all input of size n . The class FP is commonly considered to be the class of tractable or feasible first order functions contrarily, for example, to the class of functions computable in exponential time.² Indeed, a function computable by an algorithm, whose runtime is equal to 2^n , would take more than 10^{22} years to complete on a computer with clock rate at most 1 GHz on an input of size 128 (*i.e.* only 16 bytes)!

If we restrict our study to decision problems, functions whose codomain is $\{0, 1\}$, then we can define in a similar way P and, respectively, NP to be the classes of decision problems decidable in polynomial time by a deterministic machine and by a non-deterministic machine, respectively. While P is the class of decision problems that can be solved in polynomial time, NP is the class of decision problems whose solution can be checked in polynomial time. However, because of non-determinism, the set of such solutions has a cardinality exponential in (a polynomial in) the input and, consequently, it is unclear whether the two classes coincide, thus leading to the well-known open issue $P \stackrel{?}{=} NP$.

Computational complexity has been deeply studied for more than half a century by considering distinct computational models, distinct cost models, distinct functions, and problems and trying to classify and compare them leading to a rich and deep, though uncomplete, understanding of difficulties and limits of computer science [AB09, Pap03, Sip06, Zoo]. The applications of computational complexity are of high interest as understanding and finding the physical limits of computations allows the programmer to avoid them or push them using several methods (approximation algorithms, distributed architectures, heuristics, ...) and highly motivates the researcher to try to develop more efficient technologies for tomorrow (quantum theory, distributed architectures, ...).

1.1.2 Strengths and weaknesses

Although computational complexity also studies alternative computational models (RAM, circuits, ...), its strength lies in the fact that, as in computability theory, TMs remain a robust computational model for a wide variety of complexity classes. Indeed, many classes can be expressed in terms of (variants of) TMs. One can think for example of Alternating Turing Machines (ATM) for characterizing circuit complexity classes [CS76] or Oracle Turing Machines (OTM) that make possible to tame second order polynomial time complexity [KC91].

This strength is emphasized by the *Invariance Thesis* of Van Emde Boas [Boa14] stating that “*there exists a standard class of machine models, including all variants of TM and variants of RAM, where models can simulate each other with a polynomial time bounded overhead and a constant factor space bounded overhead*”.

This robustness strength is also its major weakness. While computational complexity focuses on machines, modern programmers are mostly interested in high level programming languages. It

¹The input tape is not taken into account for sublinear space classes.

²This identification of tractable functions with the complexity class FP is also known as Cobham-Edmonds thesis.

is well-known that the complexity of programs is not directly related to computational complexity as standard programmers mostly focus on asymptotic complexity rather than computational complexity. In such a framework, a simple `for` loop program guarded by the integer n will be considered to be linear in n while the corresponding machine simulating this loop using a binary representation will execute in exponential time $2^{|n|}$ in the input size, the size of the binary word representing the integer n being $|n| = \lceil \log_2(n) + 1 \rceil$.

Moreover, the *Invariance Thesis* does not hold for programs based on models like the lambda calculus or term rewriting systems as it is well-known that some reduction strategies can compute an output of exponential size in a linear number of reductions in the input size. Although the *Invariance Thesis* holds on variants of the lambda calculus based on the notion of explicit substitution and the notion of useful reduction [ADL14], it turns out that such models are in need of their own tools to be analyzed correctly.

Worst of all, computational complexity does not give a hint on how to design a program of a given complexity class. This weakness is tied to the explicit nature of the resource bound provided that does not give ways of building/certifying programs. This is particularly problematic for people interested by safety and security applications of computational complexity as it highlights the impossibility to develop automatic methods for certifying the complexity of such machines.

1.2 Implicit Computational Complexity

1.2.1 Description

The development of *Implicit Computational Complexity* (ICC) is concomitant to the will of solving all the aforementioned issues by finding static methods for analyzing automatically the complexity of “real” programming languages without explicitly referring to machines and without explicitly providing a resource bound.

ICC aims at defining criteria such that any program satisfying a given criterion will compute a function of a given complexity class. As the bound on resource consumption is implicit, the analysis is more suited to be applied in a static analysis perspective under the requirement that the criterion is not too hard to check from both a computability perspective and a complexity perspective: the programmer has no prior knowledge on the complexity of the code under analysis and wants to obtain some guarantees on it for a *safe* execution.

The aim of ICC can be summarized as follows. Given a programming language \mathcal{L} and a complexity class \mathcal{C} , find a restriction $\mathcal{R} \subseteq \mathcal{L}$ such that the following equality holds:

$$\{\llbracket \mathbf{p} \rrbracket \mid \mathbf{p} \in \mathcal{R}\} = \mathcal{C},$$

where $\llbracket \mathbf{p} \rrbracket$ is the function computed by the program \mathbf{p} .

The above equality is *extensional*, *i.e.* it deals with sets of functions rather than programs and/or machines. The semantics $\llbracket - \rrbracket$ is often omitted when it is clear from the context, for example when \mathbf{p} is a program from binary words to binary words.

The inclusion from left to right means that any function computed by a program of \mathcal{L} satisfying the criterion \mathcal{R} is in \mathcal{C} . This property is called *Soundness*. Conversely, for any function f in \mathcal{C} , there exists a program in $\mathbf{p} \in \mathcal{L}$ such that $f = \llbracket \mathbf{p} \rrbracket$ and $\mathcal{R}(\mathbf{p})$ hold. This property is called (extensional) *Completeness*.

The considered language \mathcal{L} , complexity class \mathcal{C} , and criterion \mathcal{R} are parameters of ICC characterizations that vary greatly from one work to another. As we will shortly see, the ICC literature consists of a tangle of such results where the language can range over programming languages

from functional to Object Oriented (OO), the complexity class can range from subpolynomial complexity to polynomial complexity and beyond, and where the criterion can, non exhaustively, be a type system, a syntactical restriction or a constraint based method.

1.2.2 Historical background

The paper [Cob65] is commonly accepted as the founding paper of ICC. In this paper, Cobham provides a characterization of polynomial time on function algebra that is formalized in [BC92] as follows.

Theorem 1.2.1 (Cobham's characterization of FP). *The least class of functions containing the constant function $z = 0$, the projection functions $\pi_j^n(x_1, \dots, x_n) = x_j$, the successor functions $s_i(x) = 2 \times x + i$, $i \in \{0, 1\}$, the smash function $\#(x, y) = 2^{|x| \times |y|}$ and closed under standard composition (COMP) and Bounded Recursion on Notation (BRN):*

$$\begin{aligned} \text{COMP}(f, \bar{g})(\bar{x}) &= f(\bar{g}(\bar{x})),^3 \\ \text{BRN}(f, g, h_0, h_1)(0, \bar{x}) &= f(\bar{x}), \\ \text{BRN}(f, g, h_0, h_1)(2y + i, \bar{x}) &= h_i(y, \bar{x}, \text{BRN}(f, g, h_0, h_1)(y, \bar{x})), \text{ when } 2y + i \neq 0, \\ &\text{provided that } \forall y, \bar{x}, \text{BRN}(f, g, h_0, h_1)(y, \bar{x}) \leq g(y, \bar{x}), \end{aligned}$$

is exactly FP.

Turning back to previous definition of an ICC characterization, the programming language \mathcal{L} under consideration would be a simple function algebra that can be seen as a first order equational programming language and the complexity class under consideration is FP. A program $p \in \mathcal{L}$ would pass the criterion \mathcal{R} , noted $p \in [z, \pi_j^n, s_i, \#; \text{COMP}, \text{BRN}]$, if it can be written from the base functions of Cobham's algebra $\{z, \pi_j^n, s_i, \#\}$ and using the COMP and BRN schemes as building blocks. Hence, Theorem 1.2.1 can be restated in a concise way as

$$\{\llbracket p \rrbracket \mid p \in [z, \pi_j^n, s_i, \#; \text{COMP}, \text{BRN}]\} = \text{FP}.$$

The above characterization of FP proved by Rose in [Ros87] provides a way to build new functions inductively from functions that are already in the class using a combination of the COMP and BRN schemes and is considered to be the first ICC characterization of a complexity class because it is machine independent. However it suffers from requiring an extra resource bound $g(y, \bar{x})$, whose check is very difficult, and is then not purely implicit.

This important drawback has been tackled independently by Bellantoni and Cook [BC92] and Leivant and Marion [LM93, Lei95] who have characterized FP on function algebra using a restricted version of primitive recursion scheme called *safe recursion* and *ramified recurrence* (based on a *tiering* discipline), respectively. These works mostly restrict the way recursion is allowed and provide the first pure ICC characterizations of FP and, hence, can be seen as the seminal works of the ICC community. These works have also been extended to lambda calculus [LM93] and polynomial space [LM94] as well as calculi with higher types [BNS00, Hof00], since then, these lines of works have continued to be of great and growing interest to computer scientists willing to study program complexity.

Before having an overview of the distinct existing analyses, we will try to understand and find the common mechanisms underlying most ICC works. As mentioned above, the class of functions computable in polynomial time (FP) was the main targeted class. This was primarily due to the assumed tractability of its inhabitants by Cobham-Edmonds thesis. As every programmer knows

that an exponential can be encoded simply by iterating data duplication or copy, most of the studies were performed in the spirit of preventing such bad behavior from happening. It turned out that such a simple intuition led to very distinct developments that were mostly influenced by the programming language (or paradigm) on which they were implemented. We can classify the ICC works in the literature in a somewhat arbitrary manner in distinct schools of thought depending mostly of the computational paradigm and the tool used to perform the analysis:

- function algebra and safe recursion,
- lambda calculi and light logics,
- term rewrite systems and interpretations,
- imperative programs and dataflow / control flow methods.

1.2.3 Function algebra and safe recursion

The function algebra approach consists in fixing a finite sequence of initial functions \mathcal{I} , in fixing a finite sequence of operators (or recursion schemes) \mathcal{O} mapping functions to functions, and in considering $[\mathcal{I}; \mathcal{O}]$, the smallest set of functions containing all the initial functions of \mathcal{I} and closed under the operators of \mathcal{O} . Such lines of works, for which a very detailed survey can be found in [Clo99], have provided various characterizations of well-known complexity classes, including, non-exhaustively, subpolynomial classes (Alogtime [CK93, Pit98], NC [Clo90, All91], Logspace [CT95]), polynomial time, FP [Cob65, Ros87], polynomial space, PSPACE [Clo97], and beyond (Grzegorzczuk's hierarchy [Sch69, Mül74]). Extensions to other complexity classes for counting problems have also been considered (*e.g.* #P [VW96]).

Most of the aforementioned characterizations are not *implicit* in the sense that they require the function computed by the closure of the recursion scheme operator to be bounded from above by a function of the class (as in Theorem 1.2.1). As mentioned previously, this problem was solved by Leivant and Marion [LM93] and Bellantoni and Cook [BC92]. We give below the formalism by Bellantoni and Cook where the input of a function is split between *normal* and *safe* data separated using a semicolon. In a function call $f(\bar{x}; \bar{y})$, \bar{x} is the normal data and \bar{y} is the safe data.

Theorem 1.2.2 ([BC92]). *The least class of functions containing the constant zero-ary function 0, the projection functions $\pi_j^{n,m}(x_1, \dots, x_n; x_{n+1}, \dots, x_{n+m}) = x_j$, $j \in [1, n+m]$, the successor functions $s_i(;x) = 2x + i$, $i \in \{0, 1\}$, the predecessor function $p(;2x + i) = x$, the conditional function $C(;x, y, z) = y$, if $x \bmod 2 = 0$, $C(;x, y, z) = z$ otherwise, and closed under safe composition (SCOMP)⁴ and Predicative Recursion on Notation (PRN):*

$$\begin{aligned} \text{SCOMP}(f, \bar{g}, \bar{h})(\bar{x}; \bar{y}) &= f(\bar{g}(\bar{x}; \bar{y}); \bar{h}(\bar{x}; \bar{y})) \\ \text{PRN}(f, h_0, h_1)(0, \bar{y}; \bar{z}) &= f(\bar{y}; \bar{z}) \\ \text{PRN}(f, h_0, h_1)(2x + i, \bar{y}; \bar{z}) &= h_i(x, \bar{y}; \bar{z}, \text{PRN}(f, h_0, h_1)(x, \bar{y}; \bar{z})) \text{ when } 2x + i \neq 0 \end{aligned}$$

is exactly FP. In other words, $\{\llbracket \mathbf{p} \rrbracket \mid \mathbf{p} \in [0, \pi_j^{n,m}, s_i, \mathbf{p}, \mathbf{C}; \text{SCOMP}, \text{PRN}]\} = \text{FP}$.

In [BC92], the characterization is restricted to functions with only normal data. This restriction is not necessary for our purpose as we have clearly distinguished the syntactical function from the object it computes.

⁴Here provided that $\bar{g} = g_1, \dots, g_n$, $\bar{g}(\bar{x}; \bar{y})$ stands for $g_1(\bar{x}; \bar{y}), \dots, g_n(\bar{x}; \bar{y})$.

The data duplication is controlled very cleverly in this framework: any normal data guarding a recursion cannot be used more than linearly (in its size) during this recursion as every call consumes a bit of data. Some copies can be passed to the functions in the context (h_0 and h_1) but the context cannot perform a recursion on the value computed by a recursive call (as such values are safe data), hence preventing iteration of non-linear functions (including the ones duplicating the size of their argument). To illustrate this point, consider the following function:

$$\begin{aligned} \text{double}(0;) &= 0, \\ \text{double}(2x+i;) &= s_1(; s_1(; \text{double}(x;))), \\ \text{exp}(2x+i;) &= \text{double}(\text{exp}(x;)). \end{aligned}$$

The function `double` is in $[0, \pi_j^{n,m}, s_i, p, C; \text{SCOMP}, \text{PRN}]$ and thus computes a polynomial time function (associating $s_1(; \dots s_1(; 0))$, $2n$ times, to any input of size n). However `exp` is (hopefully) not in this class as the use of `double` as contextual function requires the `exp(x;)` recursive call to be normal and this breaks the `PRN` scheme.

Predicative recursion and ramified recurrence have encountered a huge success in the community and have been altered in order to provide purely implicit characterizations of other complexity classes such as the subpolynomial classes `Alogtime` [Blo94, LM00, JDN19], `NCk` [BKMO08] and `NC` [Bel95, Lei98, JDN19], and polynomial space [Oit01]. A functional programming language based on safe recursion has also been introduced in [BCR09].

1.2.4 lambda calculus and light logics

Another very popular approach lies at the intersection of the Curry-Howard correspondence, between lambda calculus and intuitionistic logic, and the introduction of Linear Logic (LL) by Girard [Gir87]. Linear logic was introduced as a substructural logic for handling operations of duplication and erasure using new logical connectors $!$ and $?$, called the exponentials, in both classical and intuitionistic settings.

In [GSS92], Girard, Scedrov, and Scott have introduced a variant of Linear logic, named Bounded Linear Logic (BLL), introducing an annotated modality $!_x A$ that can be translated as $1 \otimes A \otimes \dots \otimes A$, with x occurrences of \otimes , ensuring that proof normalization can be performed in polynomial time and that any function in `FP` can be represented. Here variable duplication (also called contraction) is handled in a very natural way by the following principle $!_{x+y}(A \& B) \multimap !_x A \otimes !_y B$. This characterization is a cornerstone result that is not purely implicit since it suffers, as the initial function algebra studies, from the need of providing explicitly the polynomial in the type annotation. This drawback has been solved by two lines of work: Light Linear Logic (LLL) by Girard [Gir98] and Soft Linear Logic (SLL) by Lafont [Laf04].

- LLL and its affine variant, Light Affine Logic (LAL) [Asp98, AR02], provide implicit characterizations of `FP` (A proof of completeness of LAL has been provided in [Rov99]). The affine variant is obtained by adding full weakening: while a linear variable occurs exactly once in a term, an affine variable occurs at most once. The intuition of light logics is as follows: contraction is allowed for $!$ variables (*i.e.* variables that can be duplicated) but the price to pay for this is an annotation by a new modality \S , called neutral. Hence duplication is allowed but cannot be iterated. The natural deduction system for LAL taken from [BT09] is shown in Figure 1.1.

Following the encoding provided for System F , the Church representation of unary integers can be given the type $\text{Nat} = \forall \alpha. !(\alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha)$ in LAL. The unary *double* function

$$\lambda n. \lambda f. \lambda x. (n \ f \ (n \ f \ x))$$

$$\begin{array}{c}
 \frac{}{x : A \vdash x : A} \text{ (Var)} \\
 \\
 \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x. M : A \multimap B} \text{ (I}\multimap\text{)} \quad \frac{\Gamma \vdash M : A \multimap B \quad \Delta \vdash N : A}{\Gamma, \Delta \vdash M N : B} \text{ (E}\multimap\text{)} \\
 \\
 \frac{\Gamma \vdash M : A}{\Gamma, \Delta \vdash M : A} \text{ (Weak)} \quad \frac{\Gamma, x : !A, y : !A \vdash M : B}{\Gamma, z : !A \vdash M[z/x, z/y] : B} \text{ (Cntr)} \\
 \\
 \frac{\Gamma, \Delta \vdash M : A}{! \Gamma, \S \Delta \vdash M : \S A} \text{ (I}\S\text{)} \quad \frac{\Gamma \vdash N : \S A \quad \Delta, x : \S A \vdash M : B}{\Gamma, \Delta \vdash M[N/x] : B} \text{ (E}\S\text{)} \\
 \\
 \frac{x : B \vdash M : A}{!x : !B \vdash M : !A} \text{ (I!)} \quad \frac{\Gamma \vdash M : !A \quad \Delta, x : !A \vdash N : B}{\Gamma, \Delta \vdash M[N/x] : B} \text{ (E!)} \\
 \\
 \frac{\Gamma \vdash M : A \quad \alpha \notin FV(\Gamma)}{\Gamma \vdash M : \forall \alpha. A} \text{ (I}\forall\text{)} \quad \frac{\Gamma \vdash M : \forall \alpha. A}{\Gamma \vdash M : A[B/\alpha]} \text{ (E}\forall\text{)}
 \end{array}$$

where typing contexts are assumed to be disjoint and $\dagger \Gamma$, for $\dagger \in \{!, \S\}$ and $\Gamma = x_1 : A_1, \dots, x_n : A_n$, stands for $x_1 : \dagger A_1, \dots, x_n : \dagger A_n$.

Figure 1.1: Typing rules for LAL (version from [BT09])

can be typed by $!Nat \multimap \S Nat$. For any type A , the iterator

$$iter_A = \lambda f. \lambda x. \lambda n. (n \ f \ x)$$

can be given the type $!(A \multimap A) \multimap \S A \multimap Nat \multimap \S A$ and, consequently, *double* cannot be iterated for the reasons explained above.

It was shown in [Gir98] that the reduction of a LLL proof-net π to its normal form can be performed in $O((d+1)|\pi|^{2^{d+1}})$, provided that $|\pi|$ is its size and d its depth (number of nested modalities), using a reduction by levels. Consequently, the reduction can be done in polynomial time for term of fixed depth. This result does not hold for terms as demonstrated in [BT09] but was extended to any reduction strategy in [Ter01, BM10].

Theorem 1.2.3 (From [BT09]). *The class of functions representable by a typable term of LAL on binary words is exactly FP.*⁵

- SLL can be seen as a subsystem of BLL that is complete for polynomial time. The annotations of BLL are replaced by a more careful handling of variable duplication, called multiplexing, as contraction $!A \multimap (!A \otimes !A)$ and digging $!A \multimap !!A$ are not valid in such a system. Multiplexing corresponds to the validity of the following principle $!A \multimap (A \otimes \dots \otimes A)$,

⁵Each binary word $w \in \{0, 1\}^*$ is encoded by a term w of type $W = \forall \alpha. !(\alpha \multimap \alpha) \multimap !(\alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha)$. A function f is represented by the term M such that $x : W \vdash M : \S^k W$ if the proof of $\vdash M[\underline{w}/x] : \S^k W$ reduces to $\vdash f(\underline{w}) : \S^k W$, for each w .

n times, for any $n \geq 0$. Here $!$ is a marker for variable duplication. The church numerals are encoded with type $\forall\alpha.!(\alpha \multimap \alpha) \multimap (\alpha \multimap \alpha)$ and the use of duplication is thus restricted as in the case of LLL. Consequently, the degree of the polynomial is equal to the depth (number of nested modalities) of the proof π of the term M plus one (*i.e.* $O(|M|^{d(\pi)+1})$).

Theorem 1.2.4 (Adapted from [Laf04]). *The class of predicates representable by a SLL proof on binary lists is exactly P.*⁶

The above result can be extended to terms and to the complexity class FP (see [BM04]).

LLL and SLL systems have been enriched to complexity classes beyond polynomial time such as elementary time in Elementary Linear Logic (ELL) [Gir98, DJ03] and elementary lambda calculus [BBRDR18] characterizing the elementary recursive functions (functions that can be computed by a TM in time bounded by an exponential tower in the input size). Affine variants such as Elementary Affine Logic (EAL) have also been considered in [BT05]. These systems have also been adapted to the complexity class **Logspace** on a BLL variant [Sch07], the complexity class **NP** for LLL [Mau03] and SLL [GMRDR08b], and the complexity class **PSPACE** for SLL [GMRDR08a].

It is important to mention that the type system for LAL does not enjoy subject reduction and polynomiality is not preserved under β -reduction. Indeed polynomiality is preserved for proof-nets, a circuit-based representation of proofs, but not for terms. This has led to some works on the subject, the most relevant of those being the introduction of Dual Light Affine Logic (DLAL) by Baillot and Terui [BT04] that substitutes the non-linear arrow \Rightarrow to the exponential $!$ using the standard translation $(A \Rightarrow B)^* = !A^* \multimap B^*$ inspired by Barber and Plotkin’s work on Dual Intuitionistic Linear Logic [BP96]. A related type system capturing the functions computable in logarithmic space has been studied in [DLS10, DLS16].

A comparison with Bellantoni and Cook function algebra (BC) has been tackled in [MO04] by Ong and Murawski. The two authors remark that “*safe variables in LAL are not contractible (i.e. not duplicable), though normal variables are*” and manage to embed BC in LAL by restricting BC to a scheme respecting safe variables non-contractibility. This restriction comes at the price of a loss of completeness that is recovered by allowing a restricted form of recursion scheme that can be encoded in LAL on safe variables. This highlights the complementarity of the two approaches as light logics bring polymorphism and higher-order types whereas function algebra provides a more flexible treatment of variable duplication.

There are also related approaches on functional languages. In [Hof99, Hof03], Hofmann develops a type system with linearity properties, called non-size-increasing, ensuring that all definable functions are in FP. This system improves upon previous systems based on predicativity or modality restrictions by ensuring that recursive definitions can be arbitrarily nested. The key intuition is to use a resource type that “*cannot be generated out of nothing*” and, consequently, ensuring that programs cannot increase the size of their input. Again the main intuition is that program cannot compute outputs whose size is the double of the size of their input. This system in its own is sound but uncomplete for FP: it characterizes the class of functions computable in polynomial time and in linear space. Consequently, the system has to be extended with some form of predicative recursion to recover completeness over FP. The expressive power of this system was more deeply studied in [Hof02].

A last interesting and related line of work on the use of constructors for controlling the complexity of functional programs is the one by Jones [Jon01] on the expressive power of functional

⁶Here we have chosen not to specify the precise encoding as it mostly differs depending on the fragment of SLL considered [Laf04, GMRDR08b, Bai08].

languages depending on some of their properties such as the use of constructors (cons-free or non-cons free programs) or the allowed recursive calls (tail-recursion) where it is shown that second order programs are more powerful than first order programs as a decision problem is computable by a cons-free first order functional program if and only if it is in \mathbf{P} , whereas it is computable by a cons-free second order program if and only if it is in $\mathbf{EXPTIME}$.

1.2.5 Term rewrite systems and interpretations

A third line of research corresponds to the notion of (polynomial) interpretation introduced to show the termination of Term Rewrite Systems (TRS) [Lan79] and studied in [CL87, Ste92, Gie95]. Interpretation methods basically consist in assigning a weight over a well-founded domain to any expression of a given TRS so that if an expression rewrites to another, then a strict weight decrease is observed. Termination is obtained as a consequence of well-foundedness. In order to ensure such a property, it suffices that the strict weight decrease holds for any rewrite rule of the TRS and that the underlying strict order enjoys suitable properties such as closure by context and closure by substitution.

More specifically, a strictly monotonic function $[b] : \mathbb{N}^n \rightarrow \mathbb{N}$ is assigned to any symbol b of arity n of a TRS \mathcal{R} and a fresh variable $[x] \in \mathbb{N}$ is assigned to any variable x of \mathcal{R} . The assignment $[-]$ is an interpretation if its canonical extension⁷ satisfies that for each rewrite rule $l \rightarrow r$ of the TRS \mathcal{R} , $[l] > [r]$, where the strict inequality is checked for any possible assignment of its free variables. The interpretation is called polynomial if all the TRS symbols are interpreted as polynomials over the natural numbers.

It was shown in [CL92] that the functions computed by TRS admitting a polynomial interpretation have a polynomially bounded growth rate although the composition of their derivation is doubly exponentially bounded (in the size of the initial term) as demonstrated by [Geu88, Lau88]. This result was extended in [BCMT98] where a first characterization of FP is provided.

Theorem 1.2.5 (Adapted from [BCMT01]). *The set of functions computable by a TRS over unary numbers admitting a polynomial interpretation $[-]$ with the subterm property⁸ and the additivity property⁹ is exactly FP.*

Characterizations of functions computable in exponential and doubly exponential time were also provided in [BCMT98], depending on whether the successor is interpreted as a linear function of the shape $aX + b$, $a > 1, b \geq 0$, or polynomial function, respectively.

In this setting variable duplication is allowed but it comes with a price to pay for. Indeed, consider the easiest case where the interpretation ranges over the set of natural numbers \mathbb{N} . If we consider a 2-ary symbol f and a polynomial interpretation $[-] \in \mathbb{N}[X, Y]$, by subterm property, $[f](X, Y) \geq X + Y$, and, consequently, a variable duplication in a term is such that $[f(x, x)] \geq 2 \times [x]$. Hence duplication is allowed but its iteration is prevented as the interpretation of a term obtained by duplication is smaller than the interpretation of the initial term, a polynomial in the size of the term by additivity and since polynomials are closed under finite composition. Consequently, no exponential can occur under additivity assumption. To illustrate the discussion,

⁷It consists in an extension to terms defined by $[b(t_1, \dots, t_n)] = [b]([t_1], \dots, [t_n])$.

⁸The subterm property holds if for any symbol b of arity $n \geq 1$, $\forall i \leq n, \forall X_i \in \mathbb{N}, [b](\dots X_i \dots) \geq X_i$.

⁹*i.e.* the successor has an interpretation of the shape $[suc](X) = X + c, c > 0$. This property is extended to any constructor symbol c of arity strictly greater than 0 by requiring that $[c](X_1, \dots, X_n) = \sum_{i=1}^n X_i + c, c > 0$.

consider the following simple TRS computing the unary exponential as a counter-example:

$$\begin{aligned}\text{double}(0) &\rightarrow 0, \\ \text{double}(\text{suc}(x)) &\rightarrow \text{suc}(\text{suc}(\text{double}(x))), \\ \text{exp}(0) &\rightarrow \text{suc}(0), \\ \text{exp}(\text{suc}(x)) &\rightarrow \text{double}(\text{exp}(x)).\end{aligned}$$

The two rules for `double` admit the following additive and polynomial interpretation $[0] = 0$, $[\text{suc}](X) = X + 1$, $[\text{double}](X) = 3X + 1$ but it turns out that, for exponential to have an interpretation, the following inequality has to be satisfied $[\text{exp}](X + 1) > 3[\text{exp}](X) + 1$. This inequality is clearly not satisfiable by a polynomial function.

Though very interesting, this characterization was suffering from a lack of expressive power (in terms of captured programs). We will discuss this point more deeply in Section 1.3. This issue was partially solved by the introduction of the notion of quasi-interpretation [MM00, BMM01] where strictly increasing functions are replaced by increasing functions and where the strict weight decrease is replaced by a non-strict weight decrease. Consequently, contrarily to interpretations, quasi-interpretations no longer ensure a time property: termination. Quasi-interpretations rather ensure a space property under additivity requirements: the size of the computed value (and the size of each intermediate value computed during the evaluation process) is bounded by the interpretation of the initial term.

But it turns out that complexity classes can be recovered if quasi-interpretations are to be intersected with some termination techniques. Let RPO, be the set of TRS that can be shown to terminate using Recursive Path Ordering (RPO, see Subsection 2.3.3 for a definition) and where all equivalent function symbol calls are compared lexicographically (lexicographic status) or argument by argument (product status).

Theorem 1.2.6 (Adapted from [BMM11]). *The set of functions computed by TRS admitting an additive and polynomial quasi-interpretation and terminating by RPO, where all function symbols have a product status, is exactly FP.*

On the practical side, a formal library based on quasi-interpretation and RPO allowing to prove that a given program computes a function in FP has been implemented in the Coq proof assistant [FHM⁺18].

Interestingly this characterization is extended in an elegant manner to the class of functions computable in polynomial space as a lexicographic comparison on a function recursive calls preserves the polynomiality of the space needed for the evaluation of a term.

Theorem 1.2.7 (Adapted from [BMM11]). *The set of functions computed by TRS admitting an additive and polynomial quasi-interpretation and terminating by RPO, where all function symbols have either product or lexicographic status, is exactly FPSPACE.*

A characterization of `Logspace` and linear space based on quasi-interpretations is also provided in [BMM05]. This notion has also been extended in a more refined notion, called super-interpretation, that also allows us to extend the characterization to subpolynomial complexity classes such as `Alogtime` [BMP06] and NC^k [MP08b]. An exhaustive survey of the complexity class characterizations can be found in [Bon11].

For practical applications, a merge between interpretation techniques and bytecode analysis was presented in [ACGDZJ04] where the authors check properties of pre-compiled first order functional programs on a simple stack machine using quasi-interpretations and RPO. This approach

was also generalized in [ADZ04] to systems of concurrent and interactive first order functional threads.

Another related practical and successful approach is the line of work on amortized resource analysis introduced in [HJ03]. In this work, Hofmann and Jost introduced a type system using a potential-based amortized analysis to infer bounds on first order program heap space consumption. Basically, data types are annotated by potentials and type inference generates a set of linear constraints which are then solved independently by an external tool. While in the seminal papers the analysis was restricted to linear bounds in the size of the input, extensions to polynomial bounds and multi-variate polynomial bounds was designed in [HH10] and [HAH11], respectively. An extension to the heap space analysis of OO programs was performed in [HJ06, HR09, HR13] by counting the memory allocations and deallocations of Java programs and an extension to the time analysis of higher-order functional programs was performed in [JHLH10]. Related approaches using sized types on higher-order functional programs were also developed in [SvKvE07, SvEvK09, SvET13]. The amortized complexity approach is focusing on soundness results rather than on completeness results as the goal is the development of practical tools for program resource analysis as Resource-aware ML (RaML) [HAH12]. This approach is closely linked to that of polynomial interpretations as confluent TRS (or TRS with a fixed deterministic rewrite strategy) can be viewed as first order programs. Moreover, the annotations can be viewed as assignments and the constraints generated are very close to the constraints (inequalities) that can be found in polynomial interpretation methods. Amortized resource analysis has been adapted to TRS in [HM14b, HM15] and has been shown to subsume polynomial interpretations, *i.e.* if a TRS is well-typed then it admits a polynomial interpretation.

1.2.6 Imperative programs and dataflow based methods

The last approach deals with the analysis of imperative programs.

In [Jon99], a characterization of \mathbf{P} was provided in terms of read-only recursive while loop programs on binary trees.¹⁰ If recursivity is withdrawn then a characterization of $\mathbf{Logspace}$ is obtained. Such a framework has been extended to a fragment of \mathbf{C} with arrays in [KV03] and its study has been extended to the non-deterministic case in [Bon06].

In [KN04], Kristiansen and Niggel use a measure, called μ -measure, ranging over \mathbb{N} and defined on imperative loop and stack programs. This measure was previously defined by Niggel for characterizing complexity classes, including \mathbf{FP} , on functional languages [Nig00]. On the imperative paradigm, the μ -measure accounts for the number of nested loops by considering only those loops for which the variables have some interdependence, which more or less corresponds to a duplication. Kristian and Niggel show that the set of functions computed by programs of μ -measure n is exactly the class of functions \mathcal{E}^{n+2} in Grzegorzczuk's hierarchy [Grz53]. This line of work was extended in [NW06] to a more general and expressive programming language including conditional, loops and more advanced data structures such as lists or trees.

In [JK05, JK09], Jones and Kristiansen present a new approach based on a matrix typing discipline for *loop* and *while* programs.¹¹ If the program has n variables, the considered matrices are $n \times n$ square matrices whose coefficients are ranging over the finite domain $\{0, m, w, p\}$, with an underlying order $<$ satisfying $0 < m < w < p$. If the matrix M types the command c then $M_{i,j}$ encodes how the j -th variable is overwritten by the i -th variable after the execution of c . The intuition hidden under this 4-elements domain is as follows: a mwp-bound is a function of the shape $\max(\bar{X}, P(\bar{Y})) + Q(\bar{Z})$, for some polynomials P and Q . Variables in \bar{X} correspond to a

¹⁰*i.e.* These programs do not have the ability to build a binary tree.

¹¹Loops correspond to linear iteration in the size of the guard value.

maximum flow m , variables in \bar{y} to a *weak-polynomial* flow w , and variables in \bar{z} to a *polynomial* flow p (and variables that do not appear correspond to a 0 flow). Consequently, values in the typing matrix encode the way each variable is related to other variables in the mwp-bound corresponding to the command execution.

To illustrate this with an example, consider the simple loop command `loop` $X_3\{X_1 := X_1 + X_2\}$. Let X_i^0 denote the initial value stored in X_i before execution. At the end of the execution of this command, the three informal equalities $X_3 = X_3^0$, $X_2 = X_2^0$, and $X_1 = X_1^0 + X_2^0 \times X_3^0$ hold. A corresponding matrix is

$$\begin{bmatrix} m & 0 & 0 \\ p & m & 0 \\ p & 0 & m \end{bmatrix}$$

as the inequalities can be reformulated by $X_3 = \max(X_3^0)$, $X_2 = \max(X_2^0)$, and $X_1 = \max(X_1^0) + Q(X_2^0, X_3^0)$, with $Q(X, Y) = X \times Y$.¹²

This type system performs a strict control of variable duplication inside loops and while loops by requiring the closure of the matrix corresponding to the inner command to have nothing but m on its diagonal. The idea is still to prevent iteration of duplication as a command of the shape $X_k := 2 \times X_k$ cannot be typed by a matrix M such that $M_{k,k} = m$. Indeed, as $X_k = \max(P(X_k^0))$ with $P(X) = 2 \times X$, $M_{k,k}$ is at least equal to w .¹³ Consequently, this command cannot be iterated as the closure will still contain w in the diagonal.

The obtained characterization is that if a command can be typed by a given matrix then the values computed by this command are of polynomially bounded size.

A similar line of work was performed on a low level assembly-like programming language of stack machines in [Moy09]. Here a state is abstracted by its size, each instruction is abstracted by the effect it has on the size of states. The analysis allows to show termination and to study the space consumed during program execution using approaches that are inspired by size change termination principle [LJBA01] and by non-size-increasing principle [Hof02], respectively.

On the practical side, several studies have been performed. Although not always directly related to ICC, these studies mostly target a similar goal (soundness-like): finding worst case analysis for practical programs. Moreover, the techniques used are sometimes based on similar or inspired theoretical approaches. Clearly, in such a practical context, completeness is sacrificed at the price of a better tractability. We will discuss this point more deeply in Section 1.3.

The practical studies can be split in three distinct approaches:

- the SPEED tool [Gul09, GMC09] that implements a multiple counters based approach for computing symbolic bounds on the number of statements a procedure executes depending on (user-defined quantitative functions on) its inputs. The idea is simple but works well: quantitative bounds are generated using invariant generation tools and the the SPEED program has been proved to be efficient and productive on C++ Standard Template Library,
- the COSTA tool [AAG⁺07a, AAG⁺07b] that has been introduced as a new symbolic analysis and that can be used to infer termination and complexity properties of Java bytecode; The method tries to infer automatically resource upper bounds on Java bytecode using cost relations. The considered complexity properties are resource usage such as time or space [AGGZ07, AGGZ13],

¹²There is no uniqueness. See next footnote.

¹³ p is also a valid option.

- the project CerCo [AAB⁺13, AARG12] (Certified Complexity) of the European Commission FP7 that was attempting to develop a formally verified complexity preserving compiler from an expressive subset of C to some assembly code for embedded systems; the compiler is also providing certified cost annotations for programs.

Some works have also extended the studies of imperative programs to subpolynomial complexity classes. In [HS10], an imperative programming language with pointers over a graph structure called PURPLE is introduced. This language subsumes Jumping Automata on Graphs of [CR80], a particular kind of automata with a state, a graph structure and, pebbles that can move from one vertex to an adjacent vertex or that can jump directly to a vertex containing another pebble, and is shown to be strictly included in Logspace as it cannot encode undirected reachability which is known to be in Logspace as a direct consequence of Reingold’s Theorem [Rei08]. It was also shown in [HRS13] that PURPLE captures all of Logspace on locally ordered graphs.

1.3 Limits

1.3.1 Intensionality vs decidability

Most of the criteria in the literature, including the criteria presented in previous section, are extensionally complete as they consist in characterizations capturing all the functions of a given complexity class \mathcal{C} . However most of these criteria are also intensionally incomplete. It means that there exist algorithms computing a function of \mathcal{C} that can be rejected by the criterion. On the one hand, some algorithms are rejected in a highly expected way. For example, if we set $\mathcal{C} = \text{FP}$, a naive algorithm computing a function in FP using an exponential number of steps will hopefully be rejected. On the other hand, some “good” algorithms will be rejected without compromising the extensional completeness. One may think for example of the sorting function. For a given functional language \mathcal{F} , a criterion $\mathcal{K} \subseteq \mathcal{F}$ can accept a naive sorting algorithm running in time $O(n^2)$ and reject the `quicksort` algorithm (which is indeed difficult to capture as its polynomial behavior relies on the ability to show that the input array is semantically divided in two parts with no increase) so that $\{\llbracket \mathbf{p} \rrbracket \mid \mathbf{p} \in \mathcal{K}\} = \text{FP}$ holds but $\mathcal{K}(\text{quicksort})$ does not. Worst of all, most of the characterizations of FP provided in previous section do not capture the natural implementation of the `quicksort` algorithm. Historically, this kind of intensional incompleteness has been highlighted in [Col89] where Colson has shown that a function computing the minimum of two unary integers \underline{n} and \underline{m} , encoding the integers n and m , respectively, cannot be implemented by a primitive recursive algorithm on a Call-By-Name (CBN) TRS in time $O(\min(n, m))$, whereas this function can easily be implemented within such a time bound if the constraint on the algorithm is relaxed.

There have been attempts to develop intensional criteria \mathcal{R} such that the following intensional property holds:

$$\forall \mathbf{p} \in \mathcal{L}, \text{ if } \Phi(\mathbf{p}) \in \mathcal{C} \text{ then } \mathcal{R}(\mathbf{p}),$$

where Φ is a complexity measure¹⁴ à la Blum [Blu67] and under the assumption that the criterion is extensionally complete (*i.e.* $\{\llbracket \mathbf{p} \rrbracket \mid \mathbf{p} \in \mathcal{R}\} = \mathcal{C}$ holds).

For example, if $\mathcal{C} = \text{FP}$, we could set $\Phi(\mathbf{p})$ to be the running time of the program \mathbf{p} relating the size of the program input to the number of steps needed to produce the corresponding output.

¹⁴We make a slight abuse of notation as usually a Blum complexity measure maps an integer, encoding the program data and input, to a Gödel numbering of the partial computable functions by not making the distinction between the Gödel number and the function it represents.

Hence any program whose runtime is a polynomial time computable function would have to satisfy the criterion \mathcal{R} .

But clearly, such criteria come at the price of a high undecidability. Indeed, if such a criterion \mathcal{R} can be designed for decision problems of the complexity class \mathbf{P} . Then one programmer just needs to consider a program `prog` of \mathcal{L} computing a well-known NP-complete problem. Depending on whether $\mathcal{R}(\text{prog})$ holds or not, the programmer would be able to give a positive (respectively negative) answer to the \mathbf{P} vs \mathbf{NP} problem. As an illustrating example, an intensionally complete characterization of PCF programs is given in [DLG11] using a parameterized type system relying on an oracle. The undecidability lies in the oracle. For the particular case of polynomial time, it is well-known for a long time that the problem of providing an intensional characterization is Σ_2^0 -complete in the arithmetical hierarchy [Haj79].

1.3.2 Complexity of inference

A given ICC criterion can be judged on two distinct levels:

- its expressive power: has the criterion more expressive power than others?
- its inherent complexity:¹⁵ is the criterion decidable and, if so, what is its complexity?

The expressive power is very difficult to judge in general as most of the methods are incomparable. However several studies on the inherent complexity of the criteria have been performed as we will see below.

Safe recursion

For function algebras, the inherent complexity is the complexity of checking whether a given program is expressed in a safe or ramified recursion scheme. It can be easily implemented in polynomial time as it consists in checking that the program equations can be generated by the underlying function algebra grammar. The tractability of this check is clearly at the root of the most important issue of function algebra: their weak expressive power. The recursion scheme highly restricts the way programs can be written. Moreover, for a given function in \mathbf{FP} , designing a program computing the function is undecidable in general and can be very hard for some particular functions.

Light logics

The problems of type inference for LAL (EAL, respectively), consisting in checking whether a lambda term can be decorated by a type in LAL (EAL, respectively) has been studied in [Bai02] (in [CM01], respectively) under some slight restrictions (normal forms and simply typed terms)¹⁶ that have been tamed in [Bai04b].¹⁷ Consequently, type inference is decidable.

Moreover, it was shown in [ABT07], that type inference for DLAL can be checked in time polynomial in the size of the lambda term under the assumption that the System F type of the lambda term under consideration is provided.

¹⁵The inherent complexity is also called *Synthesis problem complexity*, *Derivability problem complexity* or *Type inference complexity* depending on the computational model under consideration.

¹⁶*i.e.* polymorphism is not allowed.

¹⁷The normal form hypothesis is also no longer required using a suitable notion of subtyping.

The type inference algorithm for the Soft Type Assignment (STA) of [GRDR07] based on SLL was proposed in [GRDR08] where it is shown to be decidable for simple types. It was extended in [CS12] to ML Soft Type Assignment (MLSTA), an extension to ML-like polymorphism whose type checking and type inferences are decidable. See also [CS16] for a discussion on the undecidability of the family of STA type systems when extended to System F .

These results can be summarized as follows (see [Bai08] for more details). In particular, type inference is shown to be in polynomial time on a restriction of EAL).

Type system	EAL	LAL	DLAL	STA	MLSTA
Restriction	T	T	F	T	F
Complexity	Decidable*	Decidable*	Ptime	Ptime	Decidable

Figure 1.2: Decidability and complexity of the type inference

where T is the restriction to closed simply typed terms and F is the restriction to closed typable terms in System F and where * means that the bound is known to be at least exponential.

Interpretations

For interpretations and quasi-interpretations, the synthesis problem consists in checking if a given TRS has a interpretation and quasi-interpretation, respectively. This problem was first introduced by Amadio in [Ama03, Ama05] and studied for quasi-interpretations on a space of functions including addition, maximum and coefficient over bounded rational numbers or integers: it was shown to be NP-hard on such function space and NP-complete if the coefficients are restricted to the set $\{0, 1\}$. The NP-hardness result was extended to coefficients over positive real numbers \mathbb{R}^+ in [BMMP05] and a survey of the distinct results has been provided in [Péc13].

Let $\mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+, \mathbb{Q}_d^+, \mathbb{R}^+\}$, \mathbb{Q}_d^+ being the set of rationals in \mathbb{Q}^+ of bounded representation, and define the following sets of polynomials:

- the set $\mathbb{K}[\overline{X}]$ of usual multivariate polynomials whose coefficients are in \mathbb{K} and with n variables $\overline{X} = X_1, \dots, X_n$ ranging over the field of real numbers,
- the set of $\text{MaxPoly}^{(k,d)}\{\mathbb{K}\}$ polynomials, which consists of functions obtained using constants over \mathbb{K} and arbitrary compositions of the operators $+, \times$ and \max of degree bounded by d and a max arity bounded by k ,
- and the set of $\text{MaxPlus}^{(k,d)}\{\mathbb{K}\}$ functions, which consists of functions obtained using constants over \mathbb{K} bounded by d and arbitrary compositions of the operators $+$ and \max , with a max arity bounded by k .

The obtained results can be summarized by Figure 1.3.

The first and second lines are direct consequences of Hilbert’s tenth problem undecidability, whereas the third and fourth lines are a consequence of Tarski’s quantifier elimination Theorem over real numbers [Tar51]. One important point to mention here is that the synthesis problem is exponential and not doubly exponential because the synthesis problem is more restricted than general quantifier elimination. In the first column, the symbol “–” means that the study of the synthesis problem for polynomial interpretation does not make sense whenever a *max* operator is allowed since *max* is not a strictly monotonic function.

Function space \ tool	Polynomial Interpretation	Polynomial Quasi-Interpretation
$\mathbb{K}[\overline{X}], \mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+\}$	Undecidable	Undecidable
$\text{MaxPoly}\{\mathbb{K}\}, \mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+\}$	–	Undecidable
$\mathbb{R}^+[\overline{X}]$	EXPTIME	EXPTIME
$\text{MaxPoly}^{(k,d)}\{\mathbb{R}^+\}$	–	EXPTIME
$\text{MaxPlus}^{(k,d)}\{\mathbb{K}\}, \mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+_d\}$	–	NP-complete
$\text{MaxPlus}^{(k,d)}\{\mathbb{R}^+\}$	–	NP-hard

Figure 1.3: Decidability and complexity of the synthesis problem

A last interesting remark is that we can conclude from the fourth line of Figure 1.3 that the criterion of Theorem 1.2.6, providing a characterization of FP, is decidable in EXPTIME over positive real numbers as the problem of checking whether a TRS terminates by RPO is known to be NP-complete [KN85].

Matrix based type system for imperative programs

For the imperative methods, the question of the complexity of the criterion was less central to the concerns because most of the tools are empirical and cannot be straightforwardly seen as ICC criteria as discussed in Section 1.2.6. In the work of Jones and Kristiansen [JK09], it is shown that the matrix based type system is not deterministic in the sense that several distinct matrices can be assigned to a given command. Jones and Kristiansen introduce an algorithm for solving the derivability problem, *i.e.* “given a command c does there exist a matrix M such that c has type M ?”, and show that this problem is in NP and conjecture it to be NP-complete.

1.4 Related Work

In this section, we try to relate the works on ICC with other important domains of theoretical computer science such as termination, computability, finite model theory and static analysis.

1.4.1 Termination

The links between ICC and termination are very tight. A program must terminate in order to compute a function of a given (standard) complexity class. In the other direction, a complexity class certificate ensures a termination certificate as the functions of ordinary complexity classes are total. This remark is not that interesting in the sense that complexity class certificates are much harder to infer than termination certificates. However, as termination is usually a first prerequisite for a program to compute a function of a given complexity class, it is not surprising that many ICC criteria are derived or inspired directly from termination techniques.

TRS. It has already been mentioned that polynomial interpretations have been introduced with the primary objective of demonstrating termination of TRSs [Lan79]. However, this tool was severely restricted by its intensionality as the subterm requirements are very strong and reject a lot of desirable programs. One suggestion by Hofbauer was to relax the notion of interpretation and particularly its subterm property by introducing the notion of context dependent-interpretations [Hof01]. Context dependent interpretations take an extra contextual parameter, a strictly positive real number, and are ranging over the real numbers as in the many works extending polynomial interpretation and quasi-interpretation techniques to real numbers [Der87, BMMP05, Luc05, Luc07, MP08c]. Well-foundedness is recovered by assuming that in a rewrite rule, the interpretation induces a strict decrease by at least the parameter value. In [MS08], a subclass of context dependent interpretations inducing a quadratic derivational complexity upper bound is introduced. Another interesting line of work in the use of interpretations for showing program termination are the studies generalizing polynomial interpretations to a more general codomain, the space of square matrices with integer coefficients [HW06, EWZ08]. This method was adapted to infer polynomial derivational complexity upper bounds [NZM10, Wal10, Wal15]. See also [MSW08] for a comparison between a subclass of context dependent interpretations and a subclass of triangular matrix interpretations. An adaptation to context-sensitive rewriting has also been studied in [HM14a].

The RPO based termination techniques (see [Der82, Kam80, Pla78]) were also popular in ICC as they are combined to additive quasi-interpretations to obtain characterizations of FP and FPSPACE [BMM11], as already presented in Theorem 1.2.6 and Theorem 1.2.7, respectively. The main motivations were that the Lexicographic Path Ordering (LPO), a RPO-like order where the arguments of function calls are compared lexicographically, was already known to yield multiply recursive derivation lengths [Wei95] and that the Multiset Path Ordering (MPO), a RPO-like order where recursive calls arguments are compared using multisets, was already known to yield primitive recursive derivation lengths [Hof92]. In a first attempt, the Light Lexicographic Path Ordering (LLPO) was developed in [CM00] to provide a characterization of FPSPACE. The LLPO was adapted to a notion of Polynomial Path Order (PPO*) in [AM08]. The PPO* (and its variant sPOP in [AEM15]) induces polynomial bounds on the maximal number of innermost rewrite steps and its combinations with other termination techniques such as dependency pairs [AM09], discussed in the next paragraph, and semantic labeling [Ava08] have been studied. Moreover a termination tool for automated termination, derivational complexity analysis and runtime analysis of TRS has been developed in [AM13b, AM16, AMS16].

A last very popular technique for showing the termination of TRS is the notion of Dependency Pairs (DP) of [AG00]. It consists in abstracting a given TRS as a dependency graph containing information about the function calls and control flow of the TRS and finding a strict decrease on the graph transitions with an underlying well-founded domain so that any cycle (that may correspond to a recursive call) can only occur a finite number of times. The DP method was used in [HM08, NEG13] to analyze the *innermost computational* complexity of TRS. The innermost complexity of a TRS is a partial function from natural numbers to natural numbers. It associates to a natural number n , the maximal derivation height obtained by deriving any term of the TRS of size smaller than n using an innermost reduction strategy. The partiality comes from the fact that the innermost complexity can be undefined in presence of diverging terms. This technique has been extended to Java programs and other programming languages in a tool called APROVE [GAB⁺17]. The properties related to the innermost runtime complexity are soundness results and induce that the corresponding computed function is in FP as demonstrated in [AM10b, DLM09]. Completeness is here sacrificed at the price of tractability. It is worth noticing that this soundness result has been extended to full rewriting in [AM10a]

which shows that the runtime complexity of a TRS and the runtime complexity of a TM implementation of the TRS, for decision problems and particular class of functions, are polynomially related. This is however not true in general as full rewriting and lambda calculus do not respect in general the *Invariance Thesis* [Boa14]: it means that there exists an exponential time computable function that can be implemented by a TRS with a polynomial derivation length (see also [ADL14] solution for recovering the *Invariance Thesis* in a lambda calculus setting and [ADL18] for the *Invariance Thesis* in TRS with sharing and memoization). It was proved in [MS11] that TRS whose termination can be shown using the DP method under some slight assumptions may induce a multiply recursive derivational complexity and primitive recursive upper bounds have been obtained on DP refinements based on RPO [MS09]. The ICC work presented in [MP08c] provides a sound and complete characterization of FP based on a combination of DP and polynomial interpretations. The flexibility provided by the DP method considerably improves the intensionality of this characterization with respect to the pure interpretation-based characterizations.

Lambda calculus and functional languages. For lambda calculi, types in LAL (or EAL) can be viewed as decorations of data types in System F [Gir72, Rey74], that is known to enjoy a strong normalization property. For example, unary integers in System F correspond to $\forall\alpha.(\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)$ and are decorated by $\forall\alpha.!(\alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha)$ in LAL (and by $\forall\alpha.!(\alpha \multimap \alpha) \multimap !(\alpha \multimap \alpha)$ in EAL). Other termination techniques such as size based termination [HPS96] have been adapted to functional languages to infer resource upper bounds in practice [ADL17]. In this framework, a type contains an extra information about the size of the terms it represents. A strict decrease is enforced on the sized types of recursive calls, ranging over a well-founded domain, to ensure termination [CK01, BGR08]. This also allows to recover information about the size of the intermediate results computed as in the case of additive interpretations (see [Vas08]). It is worth noticing that this approach is related to the one of [DLG11, DLP14] that combines dependent types and a linear type system to obtain type system relatively complete for PCF. Here *relatively* means that not only the complexity of the computed function is captured but also the complexity of evaluating a term. However, this approach is not a pure ICC study in the sense that, as in BLL, several external resource bounds are required and the target (PCF) is not a complexity class.

A last very popular technique for showing the termination of first order functional programs is the Size Change Principle [LJBA01] (SCP). It basically consists in abstracting a program in a call of sequences and checking that any infinite call of sequences corresponds to an infinite number of decrease on a well-founded domain. It was proved in [BA02] that the class of functions computed by size change terminating programs is the class of multiply recursive functions.

Imperative programs. On the imperative side, the works on studying the complexity of while loop programs are also related to the studies on the termination of such programs, including non exhaustively [BAG13, BAG17, BAGM12, SKvE10]. [HJP10] is also an interesting paper presenting the differences and similarities between the size change principle approach of [LJBA01] and the transition invariant approach of [PR04] that serves as a basis of the software TERMINATOR [CPR06] designed for automatically showing the termination of imperative programs.

1.4.2 Computability

ICC is also highly related to the studies on computational models and computability. Before discussing the complexity of a given function, the first question arising is whether this function

is computable by any reasonable computational model. Hence computability, as termination, is a prerequisite for complexity. After delineating the limits of computational models, researchers have focused on restricted class of computable functions. Historically, the first studies of interest mostly focused on very large classes that are now considered as untractable, *e.g.* the class of recursive functions [Pét67, Kle36]. Other examples of such classes are the class of primitive recursive functions, the class of multiple recursive functions, and classes of the Grzegorzczuk's hierarchy. Primitive recursive functions were introduced by Gödel and Kleene in [Göd31, Kle35]. Kalmar defined in [Kal43] the elementary functions by restricting the primitive recursion scheme to limited sums and products. In [Grz53], Grzegorzczuk defined the hierarchy of classes \mathcal{E}^k , obtained by closure of diagonal functions, composition and a restricted scheme of primitive recursion, and showed that \mathcal{E}^3 is exactly Kalmar's class, and, in [Rit63], Ritchie demonstrated that \mathcal{E}^2 is exactly the class of functions computable by a TM in linear space. The characterizations of these classes were obtained mostly by restricting the recursion scheme and expressive power of the underlying language and can be viewed as the corner stone approaches that led to Cobham's characterization of FP.

1.4.3 Finite model theory

The works in ICC are also related to Finite Model Theory (FMT). FMT is a branch of mathematical logic studying the relation between a (logical) language and its interpretation on finite structures. In [Fag74], Fagin showed that the properties expressible by an existential second order formula are exactly the complexity class NP. A characterization of P, known as the Immerman-Vardi Theorem, was provided (see [Saz80, Imm86] and [Var82], where other characterizations of the main complexity classes are also provided) in terms of first order logic extended by the ability to define new relations by induction expressed as a fixed-point logic over ordered structures captures P. An alternative definition was also suggested by Grädel [Grä91] in terms of second order Boolean queries in which the first order part is a universal formula in conjunctive normal form with at most one positive literal per clause. Some characterization of P and Logspace have also been studied in [Gur83] and extension to elementary recursive functions are studied in [Goe92]. See [Imm12] for a complete description of the descriptive complexity domain.

To some extent FMT can be considered as a subdomain of ICC as it provides characterizations of complexity classes without requiring resource bounds. However this is a rough approximation for two reasons. First, it is well known that the main results of classical model theory fail for finite structures and, in the other direction, the complexity results on finite models cannot be adapted to general models including functions. For example, a characterization of FP cannot be provided straightforwardly. Second, the characterizations suffer from being extensional rather than intensional as a Quantified Boolean Formula (QBF) is rather a function than a programming language or computational model.

1.4.4 Static analysis

ICC is also closely related to static program analysis. Static analysis is a subdomain of Formal Methods related to the analysis of program properties performed without actually executing the program. Techniques such as dataflow analysis, model checking, and the use of Hoare logics for correctness proofs belong to static analysis. We have already mentioned in Section 1.2.6 that methods for analyzing the complexity of imperative programs such as [JK05, JK09] were directly inspired by dataflow analysis. One of the most successful techniques in static analysis is the notion of abstract interpretation by Cousot and Cousot [CC77]. Although there is, to our knowledge,

no formal comparison between the notion of abstract interpretation and the ICC tools, it is clear that most of the semantics ICC tools can be viewed as a particular abstractions of program properties. For example, a linear type can be viewed as an abstraction of the term it corresponds to and the polynomial interpretation of an expression can be viewed as an abstraction: an upper bound on its derivation length.

It is worth noticing that most of aforementioned applied complexity analysis of imperative programming languages [Gul09, GMC09, AAG⁺07a, AAG⁺07b] are based on the use of static analysis techniques and, mostly, rely on the use of abstract interpretations to compute program invariants.

1.5 Contribution

In the first part of this introduction, we have defined the notion of ICC and described the main families of tools and criteria providing ICC results and complexity classes characterizations. We have also provided a comparison with theoretical works on computability theory, finite model theory, and well-known static analysis tools such as termination tools and abstract interpretations. We have highlighted the two main restrictions of ICC: the complexity of the inference problem and the (lack of) expressive power of the methods.

We now present our main technical and theoretical contributions to the field and will also try to place them in a more general context by not limiting this presentation to our work. The first part of this contribution, Chapter 2, will deal with the works performed on improving the intensionality/expressive power of the ICC methods. The second part, Chapter 3, will deal with the extensions of the techniques to other paradigms. The third part, Chapter 4, will deal with the extensions to infinite (including coinductive) data structures. The last part of this contribution, Chapter 5, will discuss some of the open issues and research perspectives of the domain.

Part of the works presented in this manuscript belongs to the works on ICC I have contributed to from 2004 to 2019. The underlying research has been supported by Inria Associated Team CRISTAL 2009-2012, ANR COMPLICE 2008-2014 and ANR ELICA 2014-2019 and corresponds to the following list of publications ordered by research topic. They correspond to 5 international journal publications and 14 international peer-reviewed international conferences and do not include my international workshop publications with informal proceedings and some of my publications in computer virology, on data aggregates and optimal representations, and on quantum program semantics.

The following table summarizes for each section of the contributing Chapters, the list of my corresponding publications (indexed below). The sections without publication correspond to works done by other authors.

Section	2.1	2.2	2.3	2.4	3.1	3.2	3.3	4.1	4.2	4.3	4.4
Publications			3	1,2,4,5,6,7	9,10,11,13		8,12	14,15,18	16,19	17	

- Improving the expressive power of ICC tools, Chapter 2:
 1. Guillaume Bonfante and Jean-Yves Marion and Romain Péchoux. A Characterization of Alternating Log Time by First Order Functional Programs. In *Logic for Programming, Artificial Intelligence, and Reasoning, 13th International Conference, LPAR 2006, Phnom Penh, Cambodia, November 13-17, 2006, Proceedings*, pages 90–104, 2006.

2. Jean-Yves Marion and Romain Péchoux. Resource Analysis by Sup-interpretation. In *Functional and Logic Programming, 8th International Symposium, FLOPS 2006, Fuji-Susono, Japan, April 24-26, 2006, Proceedings*, pages 163–176, 2006.
 3. Guillaume Bonfante and Jean-Yves Marion and Romain Péchoux. Quasi-interpretation Synthesis by Decomposition. In *Theoretical Aspects of Computing - ICTAC 2007, 4th International Colloquium, Macau, China, September 26-28, 2007, Proceedings*, pages 410–424, 2007.
 4. Jean-Yves Marion and Romain Péchoux. A Characterization of NC^k . In *Theory and Applications of Models of Computation, 5th International Conference, TAMC 2008, Xi'an, China, April 25-29, 2008. Proceedings*, pages 136–147, 2008.
 5. Jean-Yves Marion and Romain Péchoux. Characterizations of polynomial complexity classes with a better intensionality. In *Proceedings of the 10th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 15-17, 2008, Valencia, Spain*, pages 79–88, 2008.
 6. Jean-Yves Marion and Romain Péchoux. Sup-interpretations, a semantic method for static analysis of program resources. *ACM Trans. Comput. Log.*, 10(4), 31 pages, 2009.
 7. Romain Péchoux. Synthesis of sup-interpretations: A survey. *Theoretical Computer Science*, 467:30–52, 2013.
- Adapting ICC tools to other programming paradigms, Chapter 3:
 8. Jean-Yves Marion and Romain Péchoux. Analyzing the Implicit Computational Complexity of object-oriented programs. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2008, December 9-11, 2008, Bangalore, India*, pages 316–327, 2008.
 9. Emmanuel Hainry, Jean-Yves Marion, and Romain Péchoux. Type-based complexity analysis for fork processes. In *the 16th International Conference on Foundations of Software Science and Computation Structures, FOSSACS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, pages 305–320, Lecture Notes in Computer Science, Springer, 2013.
 10. Jean-Yves Marion and Romain Péchoux. Complexity information flow in a multi-threaded imperative language. In *the 11th Annual Conference on Theory and Applications of Models of Computation, TAMC 2014, Chennai, India, April 11-13, 2014. Proceedings*, pages 124–140, Lecture Notes in Computer Science, Springer, 2014.
 11. Emmanuel Hainry and Romain Péchoux. Objects in Polynomial Time. In *the 13th Asian Symposium on Programming Languages and Systems, APLAS 2015, Pohang, South Korea, November 30- December 2, 2015. Proceedings*, pages 387–404, Lecture Notes in Computer Science, Springer, 2015.
 12. Emmanuel Hainry and Romain Péchoux. Higher-order interpretations for higher-order complexity. In *the 21st International Conference on Logic for Programming Artificial Intelligence and Reasoning, LPAR 2017, Maun, Botswana*.
 13. Emmanuel Hainry and Romain Péchoux. A Type-Based Complexity Analysis of Object Oriented Programs. *Information and Computation*, 261:78-115, 2018.
 - Extending ICC tools to infinite and coinductive data, Chapter 4:

14. Marco Gaboardi and Romain Péchoux. Global and Local Space Properties of Stream Programs. In *Foundational and Practical Aspects of Resource Analysis - First International Workshop, FOPARA 2009, Eindhoven, The Netherlands, November 6, 2009, Revised Selected Papers*, pages 51–66, 2009.
15. Marco Gaboardi and Romain Péchoux. Upper Bounds on Stream I/O Using Semantic Interpretations. In *Computer Science Logic, 23rd international Conference, CSL 2009, 18th Annual Conference of the EACSL, Coimbra, Portugal, September 7-11, 2009. Proceedings*, pages 271–286, 2009.
16. Hugo Férée, Emmanuel Hainry, Mathieu Hoyrup, Romain Péchoux. Interpretation of Stream Programs: Characterizing Type 2 Polynomial Time Complexity. In *Algorithms and Computation - 21st International Symposium, ISAAC 2010, Jeju Island, Korea, December 15-17, 2010, Proceedings, Part I, Proceedings*, pages 291–303, Lecture Notes in Computer Science, Springer, 2010.
17. Marco Gaboardi and Romain Péchoux. Algebras and coalgebras in the light affine lambda calculus. In *the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015. Proceedings*, pages 114–126, ACM, 2015.
18. Marco Gaboardi and Romain Péchoux. On Bounding Space Usage of Streams Using Interpretation Analysis. *Science of Computer Programming*, 111(3):395–425, 2015.
19. Hugo Férée, Emmanuel Hainry, Mathieu Hoyrup, and Romain Péchoux. Characterizing polynomial time complexity of stream programs using interpretations. *Theoretical Computer Science*, 585:41–54, 2015.

Chapter 2

Towards a better intensionality

Contents

2.1	Linear logic based approaches	58
2.1.1	Light linear logic	58
2.1.2	Soft linear logic	61
2.2	Interpretations and term rewrite system	62
2.2.1	Term rewrite systems as a computational model	62
2.2.2	Interpretation methods	63
2.3	Quasi-interpretation	67
2.3.1	Motivations, definition, and basic properties	67
2.3.2	Intensional properties of quasi-interpretations	68
2.3.3	Recursive path orderings	69
2.3.4	Interpretation vs quasi-interpretation	71
2.3.5	Modularity	73
2.4	Sup-interpretation	76
2.4.1	Motivations, definition, and basic properties	76
2.4.2	Combination with the dependency pair method	78
2.4.3	Sup-interpretation vs (quasi-)interpretation	84
2.4.4	DP-interpretations for sup-interpretation synthesis	85
2.5	Summary	87

In this chapter, we address the issue of the expressive power of ICC criteria. Let us first restate the notion of an ICC criterion already discussed in Section 1.2.

Definition 2.0.1 (ICC criterion). *Given a programming language \mathcal{L} , with semantics $\llbracket - \rrbracket$ mapping every program $p \in \mathcal{L}$ to a partial function $\llbracket p \rrbracket \in \{0, 1\}^* \rightarrow \{0, 1\}^*$, and a complexity class \mathcal{C} , an ICC criterion \mathcal{R} is a subset of \mathcal{L} such that the following holds:*

$$\{\llbracket p \rrbracket \mid p \in \mathcal{R}\} = \mathcal{C}.$$

If we want to mention explicitly the language \mathcal{L} and complexity class \mathcal{C} , we write that \mathcal{R} is a $(\mathcal{L}, \mathcal{C})$ ICC criterion. For simplicity, we will sometimes write $\llbracket \mathcal{R} \rrbracket$ to denote the set of functions computed by programs $p \in \mathcal{R}$.

The expressive power of an ICC criterion is called its *intensionality* in the literature in opposition to *extensionality* that refers to functions.¹⁸ Indeed, when a program analyzer is claimed to have a better intensionality than some others, it means that the set of captured algorithms strictly includes the set of algorithms captured by the other analyzers.

Definition 2.0.2 (Intensionality). *For a fixed language \mathcal{L} and a given complexity class \mathcal{C} , let \mathcal{R} and \mathcal{S} be two $(\mathcal{L}, \mathcal{C})$ ICC criteria. \mathcal{R} has a better intensionality (or more expressive power) than \mathcal{S} , if $\mathcal{R} \subseteq \mathcal{S}$. We will write $\mathcal{R} = \mathcal{S}$ in the case where $\mathcal{R} \subseteq \mathcal{S}$ and $\mathcal{S} \subseteq \mathcal{R}$ both hold. In the case where $\mathcal{S} - \mathcal{R} \neq \emptyset$ and $\mathcal{R} - \mathcal{S} \neq \emptyset$ both hold, \mathcal{R} and \mathcal{S} are incomparable.*

Sadly, the intensionalities of most of the ICC criteria that can be found in the literature are very difficult to compare as these criteria mainly target distinct programming paradigms. We can only compare the criteria for a fixed language and a fixed complexity class. Worst of all, even for a fixed language and a fixed complexity class, ICC criteria are most of the time incomparable.

Historically the question of intensionality was introduced due to the difficulty of writing “programs” in the function algebra framework. As discussed in Section 1.3, this issue was highlighted by the work of Colson on the relative difficulty to write functions using primitive recursion [Col89]. Due to the restrictive nature of recursive calls in such a framework, function algebra were not the good candidate for improving the expressive power and the characterizations on lambda calculus, TRS, and imperative programming languages can be thought of as finding a way to improve the intensionality of function algebra.

Whereas only few works have been carried out for imperative programming languages, where completeness was often abandoned for the sake of tractability, this notion has been studied for lambda calculus and logics and studied for TRS and interpretations. We sum up most of the corresponding results in the remainder of this chapter.

In Section 2.1, we discuss the advances obtained in the main linear logic based approaches: soft linear logic and light linear logic. In Section 2.2, we recall some basic notions on TRSs. We introduce the notion of (polynomial) interpretations that was primarily used for showing termination and used later to improve the expressive power of function algebra characterizations. In Section 2.3, we discuss the notion of quasi-interpretation that improves the expressive power of interpretations. The price to pay is the loss of termination properties. We study its combinations with RPOs, a termination technique, and its modularity properties. In Section 2.4, we introduce the last notion: sup-interpretation that allows to improve the expressive power of quasi-interpretation by withdrawing the subterm property requirements. We also study its combination with the dependency pairs method.

2.1 Linear logic based approaches

2.1.1 Light linear logic

For light logics, Dual Light Affine Logic (DLAL) was introduced by Baillot and Terui in [BT04, BT09] in order to solve the main issues of LAL, *i.e.* to ensure subject reduction and to obtain a polynomial time complexity bound on β -reduction. The DLAL system includes a non-linear arrow \Rightarrow to compensate for the absence of the exponential ! and is designed to study complexity properties of terms of the pure lambda calculus

$$M, N ::= x \mid \lambda x.M \mid M N,$$

¹⁸This should not be confused with the non related philosophical concept of intentionality.

$$\begin{array}{c}
\frac{}{\Gamma; \Delta \vdash x : A} \text{ (Var)} \\
\\
\frac{\Gamma; \Delta, x : A \vdash M : B}{\Gamma; \Delta \vdash \lambda x. M : A \multimap B} \text{ (I}\multimap\text{)} \quad \frac{\Gamma; \Delta \vdash M : A \multimap B \quad \Gamma'; \Delta' \vdash N : A}{\Gamma, \Gamma'; \Delta, \Delta' \vdash M N : B} \text{ (E}\multimap\text{)} \\
\\
\frac{\Gamma, x : A; \Delta \vdash M : B}{\Gamma; \Delta \vdash \lambda x. M : A \Rightarrow B} \text{ (I}\Rightarrow\text{)} \quad \frac{\Gamma; \Delta \vdash M : A \Rightarrow B \quad \Gamma, z : C \vdash N : A}{\Gamma, z : C; \Delta \vdash M N : B} \text{ (E}\Rightarrow\text{)} \\
\\
\frac{\Gamma; \Delta \vdash M : A}{\Gamma, \Gamma'; \Delta, \Delta' \vdash M : A} \text{ (Weak)} \quad \frac{x : A, y : A, \Gamma; \Delta \vdash M : B}{z : A, \Gamma; \Delta \vdash M[z/x, z/y] : B} \text{ (Cntr)} \\
\\
\frac{\Gamma; \Delta \vdash M : A}{\Gamma; \S \Delta \vdash M : \S A} \text{ (I}\S\text{)} \quad \frac{\Gamma; \Delta \vdash N : \S A \quad \Gamma'; x : \S A, \Delta' \vdash M : B}{\Gamma, \Gamma'; \Delta, \Delta' \vdash M[N/x] : B} \text{ (E}\S\text{)} \\
\\
\frac{\Gamma; \Delta \vdash M : A \quad \Gamma'; \Delta' \vdash N : B}{\Gamma, \Gamma'; \Delta, \Delta' \vdash M \otimes N : A \otimes B} \text{ (I}\otimes\text{)} \quad \frac{\Gamma; \Delta \vdash M : A \otimes B \quad \Gamma'; \Delta', x : A, y : B \vdash N : C}{\Gamma, \Gamma'; \Delta, \Delta' \vdash \text{let } x \otimes y = M \text{ in } N : B} \text{ (E}\otimes\text{)} \\
\\
\frac{\Gamma; \Delta \vdash M : A \quad \alpha \notin FV(\Gamma) \cup FV(\Delta)}{\Gamma; \Delta \vdash M : \forall \alpha. A} \text{ (I}\forall\text{)} \quad \frac{\Gamma; \Delta \vdash M : \forall \alpha. A}{\Gamma; \Delta \vdash M : A[B/\alpha]} \text{ (E}\forall\text{)}
\end{array}$$

Figure 2.1: Typing rules for DLAL

with standard β -reduction $(\lambda x.M) N \rightarrow_{\beta} M[N/x]$, where $[N/x]$ is the standard substitution.

The types of DLAL are given by the following grammar

$$A, B ::= \alpha \mid A \multimap B \mid A \otimes B \mid A \Rightarrow B \mid \S A \mid \forall \alpha. A.$$

A judgment in DLAL is of the form $\Gamma; \Delta \vdash M : A$ and means that term M has type A under the disjoint typing environments Γ and Δ . Δ is the affine typing environment for variables, meaning that a variable in Δ occurs at most once in a term, whereas Γ is the non-linear typing environment for variables.

Typing rules for DLAL are given as a natural deduction-like system in Figure 2.1.

Example 2.1.1 (From [BT09]). *Church numerals can be encoded as $\lambda f.\lambda x.f(\dots(f x)\dots)$ in DLAL and can be given the type $\mathbf{Nat} = \forall \alpha.(\alpha \multimap \alpha) \Rightarrow \S(\alpha \multimap \alpha)$ using the following derivation.*

$$\frac{\frac{}{; x : \alpha \vdash x : \alpha} \text{(Var)} \quad \frac{}{; f_1 : \alpha \multimap \alpha \vdash f_1 : \alpha \multimap \alpha} \text{(Var)}}{; x : \alpha, f_1 : \alpha \multimap \alpha \vdash f_1 x : \alpha} \text{(E}\multimap\text{)}}{\vdots} \text{(E}\multimap\text{)} \quad \vdots$$

$$\frac{\frac{}{\vdots} \text{(E}\multimap\text{)}}{; x : \alpha, \forall i, f_i : \alpha \multimap \alpha \vdash f_n (\dots (f_1 x) \dots) : \alpha} \text{(I}\multimap\text{)}}{\frac{}{; \forall i, f_i : \alpha \multimap \alpha \vdash \lambda x.f_n (\dots (f_1 x) \dots) : \alpha \multimap \alpha} \text{(I}\S\text{)}}{\forall i, f_i : \alpha \multimap \alpha; \vdash \lambda x.f_n (\dots (f_1 x) \dots) : \S(\alpha \multimap \alpha)} \text{(Ctr)}}{\vdots} \text{(Ctr)}$$

$$\frac{\frac{}{f : \alpha \multimap \alpha; \vdash \lambda x.f (\dots (f x) \dots) : \S(\alpha \multimap \alpha)} \text{(Ctr)}}{\vdash \lambda f.\lambda x.f (\dots (f x) \dots) : (\alpha \multimap \alpha) \Rightarrow \S(\alpha \multimap \alpha)} \text{(I}\Rightarrow\text{)}}{\vdash \lambda f.\lambda x.f (\dots (f x) \dots) : \forall \alpha.(\alpha \multimap \alpha) \Rightarrow \S(\alpha \multimap \alpha)} \text{(I}\forall\text{)}$$

Using the type \mathbf{Nat} , the following types can be derived for addition and multiplication over Church's numerals.

$$\begin{aligned} \text{add} &= \lambda m.\lambda n.\lambda f.\lambda x.m f (n f x) \\ \text{add} &: \mathbf{Nat} \multimap \mathbf{Nat} \multimap \mathbf{Nat} \\ \text{mult} &= \lambda m.\lambda n.n (\lambda y.\text{add } m y) \lambda f.\lambda x.x \\ \text{mult} &: \mathbf{Nat} \Rightarrow \mathbf{Nat} \multimap \S \mathbf{Nat} \end{aligned}$$

For a typing derivation π of a term M , define its depth $d(\pi)$ to be the maximal number of premises of \S introduction rules (I \S) and the number of right-hand-side premises of \Rightarrow elimination rules (E \Rightarrow) in each branch of π .

The polynomial upper bound on the number of reduction steps can be formalized as follows.

Theorem 2.1.1 (Polynomial time soundness of DLAL [BT04]). *Given a term M of typing derivation π , for any reduction strategy, M reduces to its normal form in at most $O(|M|^{2^{d(\pi)}})$ reduction steps.*

Let W be the type for binary words defined as $W = \forall \alpha.(\alpha \multimap \alpha) \Rightarrow (\alpha \multimap \alpha) \Rightarrow \S(\alpha \multimap \alpha)$. For a binary word $w \in \{0, 1\}^*$, let \underline{w} be a term of type W encoding w . A function f is computed by the term M such that $\vdash M : \S^k W$ if, $\forall w \in \{0, 1\}^*$, $M \underline{w}$ reduces to $\underline{f(w)}$.

Completeness can be stated as follows.

Theorem 2.1.2 (Polynomial time completeness of DLAL [BT04]). *For any function f in FP, there exist a term M and a constant k such that $\vdash M : W \multimap \S^k W$ and M computes f .*

Notice that the presented version of DLAL includes a tensor product that can be trivially withdrawn if we want to compare the DLAL system with the LAL system of Figure 1.1. Moreover, the intensionality of DLAL is strictly smaller than the one of LAL.

Proposition 2.1.1.

$$\text{DLAL} \subsetneq \text{LAL}.$$

The inclusion $\text{DLAL} \subseteq \text{LAL}$ is proved using the transformation $(A \Rightarrow B)^* := !(A)^* \multimap (B)^*$ and the strict inclusion follows from the fact that some types of LAL cannot be derived in DLAL.

If one focuses on types rather than algorithms then DLAL can be shown to be equivalent to LAL, as proved in [Bai08] using the translation $(-)^{\bullet}$ from LAL to DLAL, commuting with all connectives distinct from $!$, and defined by:

$$(!A)^{\bullet} := \forall \alpha. ((A)^{\bullet} \Rightarrow \alpha) \multimap \alpha, \text{ provided that } \alpha \notin A.$$

In particular, types of the shape $A \multimap !B$ can be removed as discussed in [BT09], where it is stated that “[This removal] does not cause much loss of expressiveness in practice, since the standard decomposition of intuitionistic logic by linear logic does not use types of the form $A \multimap !B$ ”.

The expressive power of these two systems is rather modest but DLAL allows the programmer to encode natural algorithms on lists of type A , encoded as $\forall \alpha. (A \multimap \alpha \multimap \alpha) \Rightarrow \S(\alpha \multimap \alpha)$, such as insertion sort. Consequently, because of its good properties (subject reduction and polynomial time reduction), DLAL is a good candidate in terms of expressive power for light logics.

2.1.2 Soft linear logic

For soft logics, a similar issue was solved in [GRDR07]: the authors introduced a type system, named Soft Type Assignment (STA), ensuring subject-reduction and a polynomial time upper bound on β -reduction, that we present in Figure 2.2. In the $(\multimap L)$ and (cut) rules, Γ and Δ are disjoint and y is fresh. Types σ and linear types A are defined by the following grammar:

$$\begin{aligned} A, B &::= \alpha \mid \sigma \multimap A \mid \forall \alpha. A, \\ \sigma, \tau &::= A \mid !\sigma. \end{aligned}$$

Types are defined in such a way to prevent $!$ from occurring in a positive occurrence of a linear arrow.

For a typing derivation π of a term M , define its depth $d(\pi)$ to be the maximal number of applications of the rule (sp) in each branch of π .

The STA system ensures polynomial time normalization.

Theorem 2.1.3 (Polynomial time soundness of STA [GRDR07]). *Given a term M of typing derivation π , M reduces to its normal form in at most $O(|M|^{d(\pi)+1})$ reduction steps.*

Let B be the encoding $\forall \alpha. (\alpha \multimap \alpha \multimap \alpha)$ for the Boolean numbers and S be the encoding for strings of Boolean numbers.

Theorem 2.1.4 (Polynomial time completeness of STA [GRDR07]). *For any function f in FP and any polynomial P of degree n such that f is computable by a TM in time $O(P)$, there exist a term M and $k \leq n + 2$ such that $\vdash^k S \vdash M : B$ and M computes f .*

$$\begin{array}{c}
 \frac{}{x : A \vdash x : A} \text{ (Var)} \\
 \\
 \frac{\Gamma, x : \sigma \vdash M : A}{\Gamma \vdash \lambda x.M : \sigma \multimap A} \text{ (-}\circ\text{R)} \quad \frac{\Gamma \vdash M : \tau \quad x : A, \Delta \vdash N : \sigma}{\Gamma, y : \tau \multimap A \Delta \vdash N[(y M)/x] : \sigma} \text{ (-}\circ\text{L)} \\
 \\
 \frac{\Gamma \vdash M : \sigma}{\Gamma, x : A \vdash M : \sigma} \text{ (weak)} \quad \frac{\Gamma \vdash M : A \quad \Delta, x : A \vdash N : \sigma}{\Gamma, \Delta \vdash N[M/x] : \sigma} \text{ (cut)} \\
 \\
 \frac{\Gamma \vdash M : \sigma}{!\Gamma \vdash M : !\sigma} \text{ (sp)} \quad \frac{\Gamma x_1 : \tau, \dots, x_n : \tau \vdash M : \sigma}{\Gamma, x : !\tau \vdash M[x/x_1, \dots, x/x_n] : \sigma} \text{ (mult)} \\
 \\
 \frac{\Gamma \vdash M : A}{\Gamma \vdash M : \forall \alpha.A} \text{ (\forall R)} \quad \frac{\Gamma, x : A[B/\alpha] \vdash M : \sigma}{\Gamma : \forall \alpha.A \vdash M : \sigma} \text{ (\forall L)}
 \end{array}$$

Figure 2.2: Typing rules for STA

Again STA is a subsystem of SLL, which means that $\text{STA} \subsetneq \text{SLL}$. As in the case of LAL, the inclusion is strict as counter-examples can be derived.

The following result about the incomparability of STA and DLAL also holds.

Proposition 2.1.2 (From comments in [GRDR07]). *STA and DLAL are incomparable.*

2.2 Interpretations and term rewrite system

In the following sections of this chapter, we will discuss the results obtained on the expressive power of interpretation tools for studying the complexity of TRSs. For that purpose, let us first recall some basic and preliminary notions on TRSs and interpretations.

2.2.1 Term rewrite systems as a computational model

A TRS (Term Rewriting System or Term Rewrite System) is a formal system for manipulating terms over a signature by means of rules (See [BN99] for a more detailed introduction to TRS). Terms are strings of symbols consisting of a countably infinite set of variables \mathcal{X} and a first order signature Σ , a non-empty set of symbols \mathbf{b} of fixed arity $\text{ar}(\mathbf{b})$. \mathcal{X} and Σ are supposed to be disjoint. As usual, the notation $\mathcal{T}(\Sigma, \mathcal{X})$ will be used to denote the set of terms s, t, \dots of signature Σ and having variables in \mathcal{X} . Moreover, let $\mathbb{V}(t)$ denote the set of variables occurring in the term t .

A (one-hole) context $C[\diamond]$ is a term in $\mathcal{T}(\Sigma \cup \{\diamond\}, \mathcal{X})$ with exactly one occurrence of the hole \diamond , a symbol of arity 0. Given a term t and context $C[\diamond]$, let $C[t]$ denote the result of replacing the hole \diamond with the term t .

A substitution σ is a mapping from \mathcal{X} to $\mathcal{T}(\Sigma, \mathcal{X})$.

A rewrite rule for a signature Σ is a pair $l \rightarrow r$ of terms $l, r \in \mathcal{T}(\Sigma, \mathcal{X})$. A TRS is as a pair $\langle \Sigma, \mathcal{R} \rangle$ of a signature Σ and a set of rewrite rules \mathcal{R} . Using the convention of [K⁺01], for each rewrite rule $l \rightarrow r$ of a TRS, all the variables of the right-hand side r are assumed to be included in the variables of l , *i.e.* $\mathbb{V}(r) \subseteq \mathbb{V}(l)$.

A constructor TRS is a TRS $\langle \Sigma, \mathcal{R} \rangle$ in which the signature Σ can be partitioned into the disjoint union $\mathcal{C} \uplus \mathcal{F}$ of a set of function symbols \mathcal{F} and a set of constructors \mathcal{C} , such that for every rewrite rule $l \rightarrow r \in \mathcal{R}$, $l = \mathbf{f}(p_1, \dots, p_n)$ where $\mathbf{f} \in \mathcal{F}$ and where p_1, \dots, p_n are terms in $\mathcal{T}(\mathcal{C}, \mathcal{X})$, called patterns. The constructors are introduced to represent inductive data. They basically consist of a strict subset $\mathcal{C} \subset \Sigma$ of non-defined functions (a function symbol is defined if it is the root of a left-hand side term in a rule).

In what follows, we will consider *orthogonal constructor* TRS that are TRS computing functions. The notion of orthogonality requires that reduction rules of the system are all left-linear, that is each variable occurs only once on the left hand side of each rule, and there is no overlap between patterns. It is a sufficient condition to ensure that the considered TRS is confluent. It implies that an orthogonal TRS computes a fixed (partial) function, mapping input terms to an output term (and not a function mapping input terms to a set of terms in the case of non-confluent systems). This syntactic requirement could have been withdrawn in favor of a semantics restriction that would only consider TRS that compute functions. Notice that, contrarily to the completeness results, the soundness complexity results presented in the remaining sections of this chapter still hold for non-confluent TRS.

Given two terms s and t , we have that $s \rightarrow_{\mathcal{R}} t$ if there are a substitution σ , a context $C[\diamond]$ and a rule $l \rightarrow r \in \mathcal{R}$ such that $s = C[l\sigma]$ and $t = C[r\sigma]$. Throughout the manuscript, let $\rightarrow_{\mathcal{R}}^*$ ($\rightarrow_{\mathcal{R}}^+$, respectively) be the reflexive and transitive (transitive, respectively) closure of $\rightarrow_{\mathcal{R}}$. Moreover we write $s \rightarrow_{\mathcal{R}}^n t$ if n rewrite steps are performed to rewrite s to t . A TRS terminates if there is no infinite reduction through $\rightarrow_{\mathcal{R}}$.

A function symbol \mathbf{f} of arity n will define a partial function $\llbracket \mathbf{f} \rrbracket$ from constructor terms (sometimes called values) $\mathcal{T}(\mathcal{C})^n$ to $\mathcal{T}(\mathcal{C})$ by:¹⁹

$$\forall v_1, \dots, v_n \in \mathcal{T}(\mathcal{C}), \llbracket \mathbf{f} \rrbracket(v_1, \dots, v_n) = v \text{ if and only if } \mathbf{f}(v_1, \dots, v_n) \rightarrow_{\mathcal{R}}^* v \wedge v \in \mathcal{T}(\mathcal{C})$$

In this case, we write $\llbracket \mathbf{f} \rrbracket(v_1, \dots, v_n) \downarrow$ to mean that the computation ends in a normal form (constructor term). If there is no such a v (because of divergence or because evaluation cannot reach a constructor term), then $\llbracket \mathbf{f} \rrbracket(v_1, \dots, v_n) \uparrow$. A TRS computes the function f if it contains a function symbol \mathbf{f} such that $\llbracket \mathbf{f} \rrbracket = f$. Finally, we define the notion of size of a term $|e|$ which is equal to the number of symbols in e .

Example 2.2.1. Consider the following simple orthogonal TRS:

$$\begin{aligned} \text{double}(0) &\rightarrow 0 \\ \text{double}(x+1) &\rightarrow ((\text{double}(x)+1)+1) \\ \text{exp}(0) &\rightarrow \underline{1} \\ \text{exp}(x+1) &\rightarrow \text{double}(\text{exp}(x)), \end{aligned}$$

where \underline{n} is a shorthand notation for $(\dots(0+1)\dots)+1$, n times. For this particular TRS, $\mathcal{F} = \{\text{double}, \text{exp}\}$, $\mathcal{C} = \{0, +1\}$, and $x \in \mathcal{X}$. The function symbol `double` computes the total function $\llbracket \text{double} \rrbracket : \mathcal{T}(\mathcal{C}) \rightarrow \mathcal{T}(\mathcal{C})$ (over unary numbers) defined as $\llbracket \text{double} \rrbracket(\underline{n}) = \underline{2n}$. The function symbol `exp` computes the total function $\llbracket \text{exp} \rrbracket : \mathcal{T}(\mathcal{C}) \rightarrow \mathcal{T}(\mathcal{C})$ defined as $\llbracket \text{exp} \rrbracket(\underline{n}) = \underline{2^n}$.

2.2.2 Interpretation methods

Interpretation methods were introduced in [MN70, Lan79] and their methodology is as follows. Choose a suitable interpretation domain with some good properties, for example, functions over

¹⁹where $\mathcal{T}(\mathcal{C}) = \mathcal{T}(\mathcal{C}, \emptyset)$.

natural numbers and well-foundedness. Check a criterion on the program (or TRS) to ensure the required property. For example, a strict decrease (stable by context and substitution) for each reduction and termination, respectively.

The initial goal of interpretation methods was to provide a certificate of termination. Consequently, interpretation methods were mostly studied on well-founded structures such as natural numbers. In the remainder of this Section, let $(\mathbb{K}, \geq_{\mathbb{K}}^{wf})$ be an ordered set such that $\mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+, \mathbb{R}^+\}$ and let $\geq_{\mathbb{K}}^{wf}$ be a total order over \mathbb{K} whose corresponding strict order $>_{\mathbb{K}}^{wf}$ is well-founded. Moreover, let $\geq_{\mathbb{K}}$ (and $>_{\mathbb{K}}$) be the standard ordering over \mathbb{K} . We will sometimes omit the subscript \mathbb{K} when it is clear from the context.

Definition 2.2.1 (Assignment). *Given a TRS $\langle \mathcal{C} \uplus \mathcal{F}, \mathcal{R} \rangle$, an assignment $[-]_{\mathbb{K}}$ over \mathbb{K} maps:*

- every variable $x \in \mathbb{V}$ to a variable $[x]_{\mathbb{K}}$ in \mathbb{K} ,
- every symbol $b \in \mathcal{C} \uplus \mathcal{F}$ to a total function $[b]_{\mathbb{K}} : \mathbb{K}^{\text{ar}(b)} \rightarrow \mathbb{K}$.

Definition 2.2.2 (Strict monotonicity). *An assignment $[-]_{\mathbb{K}}$ is strictly monotonic if for every symbol b of arity $\text{ar}(b) > 0$, $[b]_{\mathbb{K}}$ is a monotonic function in each of its arguments, i.e. $\forall i \in [1, \text{ar}(b)]$,*

$$\forall X, Y \in \mathbb{K}, X >_{\mathbb{K}}^{wf} Y \implies [b]_{\mathbb{K}}(\dots, X_{i-1}, X, X_{i+1}, \dots) >_{\mathbb{K}}^{wf} [b]_{\mathbb{K}}(\dots, X_{i-1}, Y, X_{i+1}, \dots).$$

Definition 2.2.3 (Interpretation). *Given a TRS $\langle \mathcal{C} \uplus \mathcal{F}, \mathcal{R} \rangle$, an interpretation over \mathbb{K} is a strictly monotonic assignment $[-]_{\mathbb{K}}$ such that*

$$\forall l \rightarrow r \in \mathcal{R}, [l]_{\mathbb{K}} >_{\mathbb{K}}^{wf} [r]_{\mathbb{K}},$$

where the interpretation $[-]_{\mathbb{K}}$ is extended canonically to terms as usual and $[l]_{\mathbb{K}} >_{\mathbb{K}}^{wf} [r]_{\mathbb{K}}$ means that if $\mathbb{V}(l) = \{x_1, \dots, x_n\}$ then $\forall X_1, \dots, X_n \in \mathbb{K}, [l]_{\mathbb{K}}(X_1, \dots, X_n) >_{\mathbb{K}}^{wf} [r]_{\mathbb{K}}(X_1, \dots, X_n)$ where, for a given term t , $[t]_{\mathbb{K}}$ is the function mapping $([x_1]_{\mathbb{K}}, \dots, [x_n]_{\mathbb{K}})$ to $[t]_{\mathbb{K}}$.

As demonstrated in [Lan79], an interpretation defines a reduction ordering (i.e. a strict, stable, monotonic, and well-founded ordering).

Theorem 2.2.1. *If a TRS $\langle \mathcal{C} \uplus \mathcal{F}, \mathcal{R} \rangle$ admits an interpretation over \mathbb{K} then it terminates.*

For $\mathbb{K} = \mathbb{N}$, the above definition constitutes the basis of interpretation method as introduced in [MN70, Lan79], where $\geq_{\mathbb{N}}^{wf}$ is taken to be the standard ordering $\geq_{\mathbb{N}}$ on natural numbers.

Example 2.2.2. *The TRS of Example 2.2.1:*

$$\begin{array}{ll} \text{double}(0) \rightarrow 0 & \text{exp}(0) \rightarrow \underline{1} \\ \text{double}(x+1) \rightarrow \text{double}(x)+2 & \text{exp}(x+1) \rightarrow \text{double}(\text{exp}(x)) \end{array}$$

admits the following interpretation over \mathbb{N} :

$$\begin{array}{ll} [0]_{\mathbb{N}} = 0, & [+1]_{\mathbb{N}}(X) = X + 1 \\ [\text{double}]_{\mathbb{N}}(X) = 3 \times X + 1, & [\text{exp}]_{\mathbb{N}}(X) = 3^{2 \times X + 1}. \end{array}$$

Indeed, it is a strictly monotonic assignment and we have a strict inequality for each rule. In particular, for the last rule, we have:

$$\begin{aligned} [\text{exp}(x+1)]_{\mathbb{N}} &= 3^{2 \times [(x+1)]_{\mathbb{N}} + 1} = 3^{2(X+1)+1} = 3^{2X+3} \\ &>_{\mathbb{N}} 3 \times 3^{2X+1} + 1 = [\text{double}]_{\mathbb{N}}(3^{2X+1}) = [\text{double}]_{\mathbb{N}}([\text{exp}(x)]_{\mathbb{N}}) = [\text{double}(\text{exp}(x))]_{\mathbb{N}}. \end{aligned}$$

In the case where $\mathbb{K} = \mathbb{R}^+$ (or \mathbb{Q}^+), $\geq_{\mathbb{R}^+}^{wf}$ cannot be defined to be the natural ordering $\geq_{\mathbb{R}^+}$ on real numbers as it is well-known that $>$ is not well-founded on such a domain. This issue was solved by Lucas in [Luc07] by defining $\geq_{\mathbb{R}^+}^{wf}$ as follows:

$$\forall X, Y \in \mathbb{R}^+, X \geq_{\mathbb{R}^+}^{wf} Y \text{ if and only if } X \geq_{\mathbb{R}^+} Y + \delta,$$

for some fixed real number $\delta > 0$. Throughout the manuscript, we will sometimes use the notation \geq_{δ} or $>_{\delta}$ to refer explicitly to the order for a given δ . Interpretations over real numbers were initially introduced by Dershowitz in [Der79]. They were required to have a subterm property to compensate for the loss of well-foundedness. A interpretation has the strict subterm property if for any symbol \mathbf{b} ,

$$\forall i \in [1, ar(\mathbf{b})], \forall X_i \in \mathbb{R}^+, [\mathbf{b}]_{\mathbb{R}^+}(X_1, \dots, X_{ar(\mathbf{b})}) >_{\mathbb{R}^+} X_i.$$

It is worth mentioning that both definitions entail that the considered functions have a derivative strictly greater than 1 as the functions are strictly monotonic. However Lucas has demonstrated in [Luc07] that his notion captures more algorithms than the one in [Der79].

It is natural to restrict the space of considered functions (the interpretation codomain) to polynomials for at least two reasons. First, considering the whole space of functions is too general from a computability perspective. Second, polynomials are admitted to be a relevant set of functions in term of time and space complexity.

In what follows, let $\mathbb{K}[X_1, \dots, X_n]$ be the set of n -variable polynomials whose coefficients are in \mathbb{K} .

Definition 2.2.4 (Polynomial interpretation). *Given a TRS $\langle \mathcal{C} \uplus \mathcal{F}, \mathcal{R} \rangle$, an interpretation $[-]_{\mathbb{K}}$ is a polynomial interpretation if for every symbol $\mathbf{b} \in \mathcal{C} \uplus \mathcal{F}$, $[\mathbf{b}]_{\mathbb{K}} \in \mathbb{K}[X_1, \dots, X_{ar(\mathbf{b})}]$.*

In particular, as mentioned in Section 1.3.2, the synthesis problem is decidable in exponential time for polynomial interpretations over \mathbb{R}^+ [BMMP05] whereas Hilbert's tenth problem can be reduced to it, for $\mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+\}$, as demonstrated in [Péc13].

The synthesis problem for polynomial interpretation has been studied in [CMTU05, Luc07] where algorithms solving the constraints are described. More recently, encoding-based algorithms via SAT or SMT solving have become the state of the art for solving the synthesis problem [FGM⁺07, BLO⁺12].

However, as highlighted in [BCMT98, BCMT01], the restriction to polynomial interpretation is not enough if one wants to study polynomial time.

Example 2.2.3. *The following TRS*

$$\text{explode}(0) \rightarrow \text{nil} \qquad \text{explode}(x+1) \rightarrow c(\text{explode}(x), \text{explode}(x))$$

computes a well-balanced binary tree of size 2^n , for any unary input \underline{n} , and the corresponding computed function cannot be in FP, but it admits the following polynomial interpretation over \mathbb{N} :

$$\begin{aligned} [0]_{\mathbb{N}} &= 1, & [+1]_{\mathbb{N}}(X) &= 2X + 1, & [\text{nil}]_{\mathbb{N}} &= 0, \\ [c]_{\mathbb{N}}(X, Y) &= X + Y, & [\text{explode}]_{\mathbb{N}}(X) &= X. \end{aligned}$$

This has led to the development of another restriction, called additivity, on the interpretation of constructor symbols.

Definition 2.2.5 (Additivity). *An assignment is additive if $\forall c \in \mathcal{C}$,*

$$[c]_{\mathbb{K}}(X_1, \dots, X_{ar(c)}) = \begin{cases} \sum_{i=1}^n X_i + k_c, & \text{with } k_c \geq_{\mathbb{K}} 1, \text{ if } ar(c) > 0, \\ 0, & \text{otherwise.} \end{cases}$$

An interpretation is additive if it is an additive assignment.

The desirable property of additive interpretations is that the interpretation of a value is in Θ of its size.

Lemma 2.2.1. *Given an additive interpretation $[-]_{\mathbb{K}}$ of a TRS, there is a constant $k \in \mathbb{N} - \{0\}$ such that, for any $v \in \mathcal{T}(\mathcal{C})$:*

$$|v| \leq_{\mathbb{K}} [v]_{\mathbb{K}} \leq_{\mathbb{K}} k \times |v|.$$

Consequently, it allows to bound from above (and also below) the interpretation of a function symbol call on values as $[f(v)]_{\mathbb{K}} \leq_{\mathbb{K}} [f]_{\mathbb{K}}(k \times |v|)$, by monotonicity of interpretations.

Let $I_{add}^{poly}(\mathbb{K}, \geq_{\mathbb{K}}^{wf})$ be the set of TRS that admits an additive polynomial interpretation over \mathbb{K} .

Theorem 2.2.2. *For $\mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+, \mathbb{R}^+\}$, $[[I_{add}^{poly}(\mathbb{K}, \geq_{\mathbb{K}}^{wf})]] = \text{FP}$.²⁰*

One important question is what is the best structure, \mathbb{N}, \mathbb{Q}^+ or \mathbb{R}^+ , in term of expressive power to consider to study polynomial interpretations. As $\mathbb{N} \subsetneq \mathbb{Q}^+ \subsetneq \mathbb{R}^+$ and $\geq_{\mathbb{N}} \subsetneq \geq_1$, the counter-examples provided in [Luc07, NM10] provide the following strict inclusions.

Theorem 2.2.3. $I_{add}^{poly}(\mathbb{N}, \geq_{\mathbb{N}}) \subsetneq I_{add}^{poly}(\mathbb{Q}^+, \geq_{\mathbb{Q}^+}^{wf}) \subsetneq I_{add}^{poly}(\mathbb{R}^+, \geq_{\mathbb{R}^+}^{wf})$.

It is worth mentioning that the consideration of non-well-founded domains such as \mathbb{R}^+ or \mathbb{Q}^+ was motivated by the lack of expressive power of polynomial interpretations as the upper bound obtained on the derivational complexity, *i.e.* the maximal number of reduction steps, of programs is most of the time too rough.

Example 2.2.4. *Consider the function symbol `double` of Example 2.2.1. Finding an additive and polynomial interpretation of `double` over \mathbb{N} consists in finding a function symbol $f : \mathbb{N} \rightarrow \mathbb{N}$ satisfying the following system of inequalities:*

$$\begin{aligned} f(0) &>_{\mathbb{N}} 0, \\ \forall X \in \mathbb{N}, f(X+k) &>_{\mathbb{N}} f(X) + 2k, \text{ with } k \geq_{\mathbb{N}} 1. \end{aligned}$$

It can be shown easily that the smallest function in $\mathbb{N}[X]$ that is a solution of this system is $f(X) = 3X + 1$. This bound is rough both from a time and space point of view as the derivational complexity (maximal number of reduction steps with respect to an upper bound on the input size) of this function is linear whereas its space consumption is $2n$ on a unary input of size n .

It can be given a tighter interpretation over \mathbb{Q}^+ : $f(X) = (2 + \delta)X + \delta$, for $\delta \in \mathbb{Q}^+$ and such that $\delta < 1$.

Bonfante, Deloup and Henrot [BD10, BDH15] have generalized this result by applying the *Positivstellensatz* Theorem to interpretations over $\mathbb{R}^+[X_1, \dots, X_n]$ using the standard strict order $>$ over \mathbb{R}^+ : they have demonstrated that the strict monotonicity with respect to $>_{\delta}$ is entailed by such interpretations and they recovered a characterization of FP using additive and polynomial interpretations over real numbers.

However the extensions to real and rational numbers are not fully satisfactory as they still fail to capture a huge number of programs. One idea was to relax the well-foundedness, only keeping track of size upper bounds: this has led to the development of quasi-interpretations.

²⁰The encoding is using TRSs over unary numbers.

2.3 Quasi-interpretation

2.3.1 Motivations, definition, and basic properties

In [MM00, BMM01], the notion of quasi-interpretation is introduced by relaxing the strict decrease in the definition of interpretations. Well-foundedness and, hence, termination are lost and this allows us to consider (non-strictly) increasing functions.

Definition 2.3.1 (Monotonicity). *An assignment $[-]_{\mathbb{K}}$ is monotonic if for every symbol \mathbf{b} of arity $ar(\mathbf{b}) > 0$, $[b]_{\mathbb{K}}$ is a monotonic function in each of its arguments, i.e. $\forall i \in [1, ar(\mathbf{b})]$,*

$$\forall X, Y \in \mathbb{K}, X \geq_{\mathbb{K}} Y \implies [b]_{\mathbb{K}}(\dots, X_{i-1}, X, X_{i+1}, \dots) \geq_{\mathbb{K}} [b]_{\mathbb{K}}(\dots, X_{i-1}, Y, X_{i+1}, \dots).$$

Some important space information about the biggest size of a intermediate computed value (any value computed during the evaluation of a term) is still kept, provided that the interpretation enjoys some property, called *subterm property*, a relaxed variation of Dershowitz's subterm property for interpretation over real numbers.

Definition 2.3.2 (Subterm). *An assignment $[-]_{\mathbb{K}}$ is subterm if for every symbol \mathbf{b} of arity $ar(\mathbf{b}) > 0$:*

$$\forall i \in [1, ar(\mathbf{b})], \forall X_i \in \mathbb{K}, [b]_{\mathbb{K}}(X_1, \dots, X_n) \geq_{\mathbb{K}} X_i.$$

We are now ready to introduce the notion of quasi-interpretation that is studied more deeply in the survey [BMM11].

Definition 2.3.3 (Quasi-interpretation). *Given a TRS $\langle \mathcal{C} \uplus \mathcal{F}, \mathcal{R} \rangle$, a (polynomial or additive, respectively) quasi-interpretation is a monotonic and subterm (polynomial or additive, respectively) assignment $[-]_{\mathbb{K}}$ over \mathbb{K} satisfying:*

$$\forall l \rightarrow r \in \mathcal{R}, [l]_{\mathbb{K}} \geq_{\mathbb{K}} [r]_{\mathbb{K}},$$

where the quasi-interpretation $[-]_{\mathbb{K}}$ is extended canonically to terms.

Contrarily to (polynomial) interpretations, quasi-interpretations can deal with partial functions as they do not imply termination.

Example 2.3.1. *Consider the following TRS:*

$$\mathbf{f}(x+2) \rightarrow \mathbf{f}(x)+2, \quad \mathbf{f}(0) \rightarrow \mathbf{f}(0), \quad \mathbf{f}(1) \rightarrow 1.$$

The function \mathbf{f} computes the identity function on odd numbers whereas it diverges on even numbers. However it admits the trivial polynomial and additive quasi-interpretation $[-]_{\mathbb{K}}$ defined by $[\mathbf{f}]_{\mathbb{K}}(X) = X$, $[+1]_{\mathbb{K}}(X) = X + 1$, and $[0]_{\mathbb{K}} = 0$.

Indeed, for the first rule, we check:

$$\begin{aligned} [\mathbf{f}(x+2)]_{\mathbb{K}} &= [\mathbf{f}]_{\mathbb{K}}([(x+1)+1]_{\mathbb{K}}) = X + 2 \\ &\geq_{\mathbb{K}} [\mathbf{f}(x)]_{\mathbb{K}} + 2 = [(\mathbf{f}(x)+1)+1]_{\mathbb{K}}. \end{aligned}$$

For the second, rule we clearly have $[\mathbf{f}(0)]_{\mathbb{K}} \geq [\mathbf{f}(0)]_{\mathbb{K}}$ and, for the last rule, we have $[\mathbf{f}(1)]_{\mathbb{K}} = [\mathbf{f}]_{\mathbb{K}}([1]_{\mathbb{K}}) \geq [1]_{\mathbb{K}}$.

2.3.2 Intensional properties of quasi-interpretations

By monotonicity and subterm properties, quasi-interpretations allow us to bound the interpretation of any intermediate value.

Lemma 2.3.1. *Given a TRS admitting a quasi-interpretation $[-]_{\mathbb{K}}$ over \mathbb{K} , for any term t , for any value $v \in \mathcal{T}(\mathcal{C})$, and any context $C[\diamond]$ such that $t \rightarrow_{\mathcal{R}}^* C[v]$,*

$$[t]_{\mathbb{K}} \geq_{\mathbb{K}} [v]_{\mathbb{K}}.$$

Hence, in the case of a polynomial and additive quasi-interpretation, it turns out that any intermediate value u computed from a term $\mathbf{f}(v)$ has size bounded polynomially in the input size (i.e. $|u| \leq_{\mathbb{K}} [\mathbf{f}]_{\mathbb{K}}(k \times |v|)$ for some polynomial $[\mathbf{f}]_{\mathbb{K}}$) by combining the above Lemma with Lemma 2.2.1. The above Lemma was also satisfied by polynomial interpretations over \mathbb{N} as a strictly increasing polynomial has to satisfy the subterm property.

Moreover, because of a relaxed monotonicity, the obtained quasi-interpretations are tighter to the effective space complexity of the studied TRS.

Example 2.3.2. *The program of Example 2.2.2*

$$\begin{aligned} \text{double}(0) &\rightarrow 0 \\ \text{double}(x+1) &\rightarrow \text{double}(x)+2 \end{aligned}$$

admits the following additive and polynomial quasi-interpretation over \mathbb{N} :

$$\begin{aligned} [0]_{\mathbb{N}} &= 0, & [+1]_{\mathbb{N}}(X) &= X + 1, \\ [\text{double}]_{\mathbb{N}}(X) &= 2 \times X, \end{aligned}$$

whereas we have already shown in Example 2.2.4 that the smallest admissible interpretation over \mathbb{N} is such that $[\text{double}]_{\mathbb{N}}(X) = 3 \times X + 1$.

Let $QI_{add}^{poly}(\mathbb{K}, \geq_{\mathbb{K}})$ be the set of function symbols whose TRS admits an additive and polynomial interpretation over \mathbb{K} . We obtain a first intensionality result as the set of programs admitting an additive and polynomial interpretations is strictly included in the set of programs admitting an additive and polynomial quasi-interpretation:²¹

Theorem 2.3.1. *For $\mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+, \mathbb{R}^+\}$, $I_{add}^{poly}(\mathbb{K}, \geq_{\mathbb{K}}^{wf}) \subsetneq QI_{add}^{poly}(\mathbb{K}, \geq_{\mathbb{K}})$.*

This result relies on the fact that strictly monotonic polynomials have the subterm property. It is worth mentioning that in the above Theorem the order differs for interpretations and quasi-interpretations on non-well-founded domains. For example, if $\mathbb{K} = \mathbb{R}^+$ then the result can be stated as follows $\forall \delta > 0$, $I_{add}^{poly}(\mathbb{R}^+, \geq_{\delta}) \subsetneq QI_{add}^{poly}(\mathbb{R}^+, \geq_{\mathbb{R}^+})$.

As quasi-interpretations do not focus on termination, domains such as $(\mathbb{R}^+, \geq_{\mathbb{R}^+})$ or $(\mathbb{Q}^+, \geq_{\mathbb{Q}^+})$ are good candidates to define quasi-interpretations. Moreover, non-strictly monotonic, subterm, and polynomially bounded functions such as the maximum can be added to the interpretation domain in order to increase the expressive power without breaking the soundness. This has led to the definition and study of the class of max-polynomials $\text{MaxPoly}\{\mathbb{K}\}$, consisting in the class of functions containing all polynomials of $\mathbb{K}[X_1, \dots, X_n]$ and closed under the *max* operation. The synthesis problem remains decidable in exponential time $\text{MaxPoly}\{\mathbb{R}^+\}$, for bounded max

²¹This result remains true if additivity and polynomiality properties are withdrawn.

degree (number of operands) k , as any inequality involving the max operator can be transformed into at most k^2 inequalities with no max (see [BMMP05, Péc13] for a more detailed treatment).

Let $QI_{add}^{maxpoly}(\mathbb{K}, \geq_{\mathbb{K}})$ be the set of function symbols whose TRS admits an additive max-polynomial and additive interpretation over \mathbb{K} .

Theorem 2.3.2. For $\mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+, \mathbb{R}^+\}$, $QI_{add}^{poly}(\mathbb{K}, \geq_{\mathbb{K}}) \subsetneq QI_{add}^{maxpoly}(\mathbb{K}, \geq_{\mathbb{K}})$.

The interest of considering quasi-interpretations over the reals does not only rely on the decidability of their synthesis contrarily to quasi-interpretations over \mathbb{N} or \mathbb{Q}^+ . Indeed, we have a result analog to Theorem 2.2.3 over MaxPoly quasi-interpretations. It states that there exist programs that do not have any quasi-interpretation over $\text{MaxPoly}\{\mathbb{Q}\}$ and, *a fortiori* $\text{MaxPoly}\{\mathbb{N}\}$, but that admit a quasi-interpretation over $\text{MaxPoly}\{\mathbb{R}^+\}$. This was demonstrated by contradiction in [Péc13] by introducing a TRS enforcing a multiplicative coefficient a of the interpretation to satisfy the equation $a^2 = 2$.

Theorem 2.3.3. $QI_{add}^{maxpoly}(\mathbb{N}, \geq_{\mathbb{N}}) \subseteq QI_{add}^{maxpoly}(\mathbb{Q}^+, \geq_{\mathbb{Q}^+}) \subsetneq QI_{add}^{maxpoly}(\mathbb{R}^+, \geq_{\mathbb{R}^+})$.

In the above Theorem, it is not known whether the first inclusion is strict or not.

2.3.3 Recursive path orderings

Theorem 2.3.3 is not a characterization. Indeed $QI_{add}^{maxpoly}(\mathbb{K}, \geq_{\mathbb{K}})$ is not a complexity class as it contains non-terminating programs. For capturing a complexity class, quasi-interpretation have to be complemented by some termination ordering in order to recover complexity results. This was done in [MM00, BMM01] and surveyed in [BMM11] by combining quasi-interpretations and recursive path ordering \prec_{rpo} (and its reflexive closure \preceq_{rpo}) that we define in Figure 2.3 with respect to a TRS $\langle \mathcal{C} \uplus \mathcal{F}, \mathcal{R} \rangle$. Let $\prec_{\mathcal{F}}$ be a strict ordering on \mathcal{F} and $\approx_{\mathcal{F}}$ a compatible equivalence relation. Define $\preceq_{\mathcal{F}}$ by $\preceq_{\mathcal{F}} = \prec_{\mathcal{F}} \cup \approx_{\mathcal{F}}$. Let st be a mapping from \mathcal{F} to $\{p, l\}$ compatible with $\preceq_{\mathcal{F}}$, i.e. $\mathbf{f} \approx_{\mathcal{F}} \mathbf{g}$ implies that $st(\mathbf{f}) = st(\mathbf{g})$. st provides information on how to compare the arguments of two equivalent function symbols. This comparison can be product p (i.e. component by component) or lexicographic l .

A TRS $\langle \mathcal{C} \uplus \mathcal{F}, \mathcal{R} \rangle$ is oriented by \prec_{rpo} if there is a status st and a precedence $\prec_{\mathcal{F}}$ such that for each rule $t \rightarrow_{\mathcal{R}} s \in \langle \mathcal{C} \uplus \mathcal{F}, \mathcal{R} \rangle$, $s \prec_{rpo} t$. For $A \subseteq \{p, l\}$, let RPO^A be the set of TRSs that can be oriented by \prec_{rpo} in such a way that all function symbols have a status in A . As rule (Lexi) is implied by rule (Prod) in Figure 2.3, but not the converse, it holds that $RPO^{\{p\}} \subsetneq RPO^{\{l\}}$ and, consequently, $RPO^{\{p, l\}} = RPO^{\{l\}}$.

Example 2.3.3. Consider the following TRS

$$\begin{aligned} \text{concat}(\mathbf{i}(\mathbf{x}), \mathbf{y}) &\rightarrow \mathbf{i}(\text{concat}(\mathbf{x}, \mathbf{y})), \mathbf{i} \in \{0, 1\}, \\ \text{concat}(\epsilon, \mathbf{y}) &\rightarrow \mathbf{y}, \end{aligned}$$

concatenating two binary words given as input. It can be oriented by \prec_{rpo} in

$$\frac{\frac{\frac{\mathbf{x} = \mathbf{x}}{\mathbf{x} \prec_{rpo} \mathbf{i}(\mathbf{x})} \text{ (SubI)} \quad \frac{\mathbf{y} = \mathbf{y}}{\mathbf{y} \preceq_{rpo} \mathbf{y}} \text{ (RefI)}}{\{\mathbf{x}, \mathbf{y}\} \prec_{rpo}^p \{\mathbf{i}(\mathbf{x}), \mathbf{y}\}} \text{ (Prod)} \quad \frac{\frac{\mathbf{x} = \mathbf{x}}{\mathbf{x} \prec_{rpo} \mathbf{i}(\mathbf{x})} \text{ (SubI)}}{\mathbf{x} \prec_{rpo} \text{concat}(\mathbf{i}(\mathbf{x}), \mathbf{y})} \text{ (Sub)} \quad \frac{\mathbf{y} = \mathbf{y}}{\mathbf{y} \prec_{rpo} \text{concat}(\mathbf{i}(\mathbf{x}), \mathbf{y})} \text{ (SubI)}}{\frac{\text{concat}(\mathbf{x}, \mathbf{y}) \prec_{rpo} \text{concat}(\mathbf{i}(\mathbf{x}), \mathbf{y})}{\mathbf{i}(\text{concat}(\mathbf{x}, \mathbf{y})) \prec_{rpo} \text{concat}(\mathbf{i}(\mathbf{x}), \mathbf{y})} \text{ (Cons)}} \text{ (FEq)}$$

$$\begin{array}{c}
\frac{s = t \quad \mathbf{b} \in \mathcal{C} \uplus \mathcal{F}}{s \prec_{rpo} \mathbf{b}(\dots, t, \dots)} \text{ (SubI)} \quad \frac{s \prec_{rpo} t \quad \mathbf{b} \in \mathcal{C} \uplus \mathcal{F}}{s \prec_{rpo} \mathbf{b}(\dots, t, \dots)} \text{ (Sub)} \\
\\
\frac{s = t}{s \preceq_{rpo} t} \text{ (RefI)} \quad \frac{\mathbf{f} \in \mathcal{F} \quad \mathbf{c} \in \mathcal{C} \quad \forall i, s_i \prec_{rpo} \mathbf{f}(t_1, \dots, t_n)}{\mathbf{c}(s_1, \dots, s_m) \prec_{rpo} \mathbf{f}(t_1, \dots, t_n)} \text{ (Cons)} \\
\\
\frac{s \prec_{rpo} t}{s \preceq_{rpo} t} \text{ (Ref)} \quad \frac{\mathbf{f}, \mathbf{g} \in \mathcal{F} \quad \mathbf{g} \prec_{\mathcal{F}} \mathbf{f} \quad \forall i, s_i \prec_{rpo} \mathbf{f}(t_1, \dots, t_n)}{\mathbf{g}(s_1, \dots, s_n) \prec_{rpo} \mathbf{f}(t_1, \dots, t_n)} \text{ (FLess)} \\
\\
\frac{\mathbf{f}, \mathbf{g} \in \mathcal{F} \quad \mathbf{g} \approx_{\mathcal{F}} \mathbf{f} \quad \forall i, s_i \prec_{rpo} \mathbf{f}(t_1, \dots, t_n) \quad \{s_1, \dots, s_n\} \prec_{rpo}^{st(\mathbf{f})} \{t_1, \dots, t_n\}}{\mathbf{g}(s_1, \dots, s_n) \prec_{rpo} \mathbf{f}(t_1, \dots, t_n)} \text{ (FEq)} \\
\\
\frac{\forall i, s_i \preceq_{rpo} t_i \quad \exists j, s_j \prec_{rpo} t_j}{\{s_1, \dots, s_n\} \prec_{rpo}^p \{t_1, \dots, t_n\}} \text{ (Prod)} \quad \frac{\exists j, \forall i < j, s_i \preceq_{rpo} t_i \quad s_j \prec_{rpo} t_j}{\{s_1, \dots, s_n\} \prec_{rpo}^l \{t_1, \dots, t_n\}} \text{ (Lexi)}
\end{array}$$

Figure 2.3: Recursive path ordering

and in

$$\frac{\mathbf{y} = \mathbf{y}}{\mathbf{y} \prec_{rpo} \text{concat}(\epsilon, \mathbf{y})} \text{ (SubI)}.$$

Moreover all function symbols have a product status. Consequently, this TRS is in $RPO^{\{p\}}$.

Example 2.3.4. Consider the following TRS

$$\begin{array}{l}
\text{step}(\mathbf{i}(\mathbf{x}), \mathbf{y}) \rightarrow \text{step}(\mathbf{x}, \mathbf{i}(\mathbf{y})), \mathbf{i} \in \{0, 1\}, \\
\text{step}(\epsilon, \mathbf{y}) \rightarrow \mathbf{y}, \\
\text{reverse}(\mathbf{x}) \rightarrow \text{step}(\mathbf{x}, \epsilon),
\end{array}$$

computing the reverse binary word of its input. It can be oriented by \prec_{rpo} using the precedence step $\prec_{\mathcal{F}}$ reverse in

$$\frac{\frac{\mathbf{x} = \mathbf{x}}{\mathbf{x} \prec_{rpo} \text{reverse}(\mathbf{x})} \text{ (SubI)} \quad \frac{}{\epsilon \prec_{rpo} \text{reverse}(\mathbf{x})} \text{ (Cons)}}{\text{step}(\mathbf{x}, \epsilon) \prec_{rpo} \text{reverse}(\mathbf{x})} \text{ (FLess)}, \\
\frac{\mathbf{y} = \mathbf{y}}{\mathbf{y} \prec_{rpo} \text{step}(\epsilon, \mathbf{y})} \text{ (SubI)}$$

and in

$$\frac{\frac{\mathbf{x} = \mathbf{x}}{\mathbf{x} \prec_{rpo} \mathbf{i}(\mathbf{x})} \text{ (SubI)} \quad \frac{\mathbf{x} = \mathbf{x}}{\mathbf{x} \prec_{rpo} \text{step}(\mathbf{i}(\mathbf{x}), \mathbf{y})} \text{ (SubI)}}{\frac{\{\mathbf{x}, \mathbf{i}(\mathbf{y})\} \prec_{rpo}^1 \{\mathbf{i}(\mathbf{x}), \mathbf{y}\}}{\mathbf{i}(\mathbf{x}) \prec_{rpo} \text{step}(\mathbf{i}(\mathbf{x}), \mathbf{y})} \text{ (Lexi)} \quad \frac{\mathbf{y} = \mathbf{y}}{\mathbf{y} \prec_{rpo} \text{step}(\mathbf{i}(\mathbf{x}), \mathbf{y})} \text{ (SubI)}}{\text{step}(\mathbf{x}, \mathbf{i}(\mathbf{y})) \prec_{rpo} \text{step}(\mathbf{i}(\mathbf{x}), \mathbf{y})} \text{ (FEq)}.$$

Moreover, the status of **step** is enforced to be lexicographic, i.e. to satisfy $st(\text{step}) = l$, as it does not hold that $\mathbf{i}(\mathbf{y}) \prec_{rpo} \mathbf{y}$. Consequently, this TRS is in $RPO^{\{l\}}$ but not in $RPO^{\{p\}}$.

Now we are ready to present the characterizations of FP and FPSPACE presented in [BMM11]. They consist in slight variations, improvements or simplifications of the results of [MM00, BMM01].

Theorem 2.3.4 ([BMM11]). *For $\mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+, \mathbb{R}^+\}$, we have:*

- $\llbracket QI_{add}^{maxpoly}(\mathbb{K}, \geq_{\mathbb{K}}) \cap RPO\{p\} \rrbracket = \text{FP}$,
- $\llbracket QI_{add}^{maxpoly}(\mathbb{K}, \geq_{\mathbb{K}}) \cap RPO\{l\} \rrbracket = \text{FPSPACE}$.

The characterization of FPSPACE is a direct application of Lemma 2.3.1 in the case of additive and (max-)polynomial interpretations. Indeed, in such a case, any value (and term) computed during the evaluation process has size bounded polynomially by the input size. The characterization of FP is a bit more subtle and needs the application of a memoization technique to show that only a polynomial number (in the input size) of recursive calls can be performed for a given function symbol.

Example 2.3.5. *The TRS of Example 2.3.3 admits the following additive and polynomial quasi-interpretation:*

$$\begin{aligned} [\text{concat}]_{\mathbb{N}}(X, Y) &= X + Y, \\ [\text{i}]_{\mathbb{N}}(X) &= X + 1, \quad \text{i} \in \{0, 1\}, \\ [\epsilon]_{\mathbb{N}} &= 0. \end{aligned}$$

As it is in $RPO\{p\}$, we can conclude, using Theorem 2.3.4, that the computed function is in FP.

Example 2.3.6. *The TRS of Example 2.3.4 admits the following additive and polynomial quasi-interpretation:*

$$\begin{aligned} [\text{step}]_{\mathbb{N}}(X, Y) &= X + Y, \\ [\text{i}]_{\mathbb{N}}(X) &= X + 1, \quad \text{i} \in \{0, 1\}, \\ [\epsilon]_{\mathbb{N}} &= 0, \\ [\text{reverse}]_{\mathbb{N}}(X) &= X. \end{aligned}$$

As it is in $RPO\{l\}$, we can conclude, using Theorem 2.3.4, that the computed function is in FPSPACE. Notice that this result does not provide a good precision on the function complexity. Some other characterizations have been developed in [BMM11]. They manage in particular to show that $\llbracket \text{reverse} \rrbracket$ belongs to FP.

2.3.4 Interpretation vs quasi-interpretation

By Theorem 2.3.1, quasi-interpretations have strictly more expressive power than interpretations. However, the characterization of Theorem 2.3.4 does not provide more intensionality than the one of Theorem 2.2.2 as illustrated by the following counter-example.

Example 2.3.7. *The TRS, computing the zero function, and consisting of the two rules*

$$\begin{aligned} \mathbf{f}(x+1) &\rightarrow \mathbf{f}(\mathbf{f}(x)), \\ \mathbf{f}(0) &\rightarrow 0. \end{aligned}$$

This TRS has the following polynomial and additive interpretation over \mathbb{N}

$$\begin{aligned} [\mathbf{f}]_{\mathbb{N}}(X) &= X + 1, \\ [+1]_{\mathbb{N}}(X) &= X + 2, \\ [0]_{\mathbb{N}} &= 0. \end{aligned}$$

It does not terminate by \prec_{rpo} as, for the first rule, it cannot hold that $\{\mathbf{f}(\mathbf{x})\} \prec_{rpo}^P \{\mathbf{x} + 1\}$.

The converse also holds, *i.e.* the characterization of Theorem 2.2.2 does not provide more intensionality than the one of Theorem 2.3.4, as illustrated by the following counter-example.

Example 2.3.8. The TRS, computing the zero function and consisting of the three rules

$$\begin{aligned} \mathbf{f}(\mathbf{x} + 1) &\rightarrow \mathbf{g}(\mathbf{f}(\mathbf{x}), \mathbf{f}(\mathbf{x})), \\ \mathbf{g}(\mathbf{x}, \mathbf{y}) &\rightarrow \mathbf{x}, \\ \mathbf{f}(0) &\rightarrow 0, \end{aligned}$$

has the following polynomial and additive quasi-interpretation over \mathbb{N}

$$\begin{aligned} [\mathbf{f}]_{\mathbb{N}}(X) &= X, \\ [\mathbf{g}]_{\mathbb{N}}(X, Y) &= \max(X, Y), \\ [+1]_{\mathbb{N}}(X) &= X + 1, \\ [0]_{\mathbb{N}} &= 0, \end{aligned}$$

and can be oriented by \prec_{rpo} only using product status. However it has no polynomial and additive interpretation as, for the first rule, an interpretation should satisfy the following inequality:

$$[\mathbf{f}]_{\mathbb{N}}(X + k) > [\mathbf{g}(\mathbf{f}(\mathbf{x}), \mathbf{f}(\mathbf{x}))]_{\mathbb{N}} \geq 2 \times [\mathbf{f}]_{\mathbb{N}}(X),$$

which is not satisfiable by any polynomial $[\mathbf{f}]_{\mathbb{N}} \in \mathbb{N}[X]$.

From the two above counter-example, we can deduce the following intensional result.

Theorem 2.3.5. For $\mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+, \mathbb{R}^+\}$, $I_{add}^{poly}(\mathbb{K}, \geq_{\mathbb{K}})$ and $(QI_{add}^{poly}(\mathbb{K}, \geq_{\mathbb{K}}) \cap RPO^{\{p\}})$ are incomparable.

RPO is still a powerful technique as Hofbauer has demonstrated in [Hof92] that the set of functions computed by $RPO^{\{p\}}$ programs is the set of primitive recursive functions. However its combination with quasi-interpretations is restrictive and it turns out that the two characterizations of FP using interpretations (Theorem 2.2.2) and using quasi-interpretations with RPO (Theorem 2.3.4) are very close from an intensional point of view as the counter-examples used for showing incomparability mostly rely on the inherent weakness of each technique: the inability to compute a sublinear function such as \max on the interpretation side, the inability to nest equivalent function calls on the RPO side.

It is worth mentioning that several works have been developed to improve the expressive power of quasi-interpretation method. In [BDLM12], the blind abstraction of a program on binary lists has been defined as the possibly non-deterministic program obtained by replacing lists with their lengths encoded as unary numbers. A program is blindly polynomial if its blind abstraction terminates in polynomial time. This notion is combined with quasi-interpretation to improve the expressive power of previous characterizations.

Another important point to mention is that we have presented a version of RPO using the Product Path Ordering (PPO). The expressive power of the method can be improved using Multiset Path Ordering (MPO) as suggested in [Bon11]. It differs in the sense that the arguments of recursive calls are compared using a multiset based comparison rather than a product one (*i.e.* component by component), allowing the treatment of rules of the shape $\mathbf{f}(\mathbf{x}+1, \mathbf{y}) \rightarrow \mathbf{f}(\mathbf{x}, \mathbf{x})$.

2.3.5 Modularity

The paper [BMP07] has suggested another approach for trying to improve the expressive power of quasi-interpretations: modularity. Modularity is a well know notion for TRSs that has been deeply studied for termination [Gra94, KO92]. Applied to our context, the main idea is to divide a TRS $\langle \mathcal{C} \uplus \mathcal{F}, \mathcal{R} \rangle$ into sub-TRSs $\langle \mathcal{C}_1 \uplus \mathcal{F}_1, \mathcal{R}_1 \rangle$ and $\langle \mathcal{C}_2 \uplus \mathcal{F}_2, \mathcal{R}_2 \rangle$ and try to search for quasi-interpretations for each of them. Among the distinct ways a program can be split, there are three main possibilities studied by the rewriting community:

- disjoint union \uplus , whenever $\langle \mathcal{C}_1 \uplus \mathcal{F}_1, \mathcal{R}_1 \rangle$ and $\langle \mathcal{C}_2 \uplus \mathcal{F}_2, \mathcal{R}_2 \rangle$ do not share any common symbol,
- constructor-sharing union \sqcup , whenever only constructor symbols in \mathcal{C}_1 and \mathcal{C}_2 are shared,
- hierarchical union \sqsubset , whenever some constructor symbols of \mathcal{C}_1 are function symbols of \mathcal{F}_2 .

In the case of hierarchical union, constructor symbols can be shared. Consequently, the constructor-sharing union can be seen a restricted (commutative) case of hierarchical union.

Now we can define the standard notion of modularity.

Definition 2.3.4. *A property R (a decision problem mapping each TRS to true or false) is modular for the union in $X \in \{\uplus, \sqcup, \sqsubset\}$, if for all TRSs \mathcal{p}_1 and \mathcal{p}_2 such that $\mathcal{p}_1 X \mathcal{p}_2$ is defined, if $R(\mathcal{p}_1)$ and $R(\mathcal{p}_2)$ then $R(\mathcal{p}_1 X \mathcal{p}_2)$.*

The following modularity results hold for (quasi-)interpretations.

Proposition 2.3.1. *The property of having a (polynomial) additive quasi-interpretation is:*

- modular for the disjoint union,
- not modular for the constructor-sharing union and the hierarchical union.

The modularity of the disjoint union is straightforward. We give one example illustrating the non-modularity of additive (quasi-)interpretations for the constructor-sharing union.

Example 2.3.9. *Consider the TRSs $\langle \mathcal{C}_1 \uplus \mathcal{F}_1, \mathcal{R}_1 \rangle$ and $\langle \mathcal{C}_2 \uplus \mathcal{F}_2, \mathcal{R}_2 \rangle$ defined by:*

$$\mathcal{R}_1 = \begin{cases} \mathbf{f}(\mathbf{a}(\mathbf{x})) & \rightarrow \mathbf{f}(\mathbf{f}(\mathbf{x})) \\ \mathbf{f}(\mathbf{b}(\mathbf{x})) & \rightarrow \mathbf{a}(\mathbf{a}(\mathbf{f}(\mathbf{x}))) \end{cases} \quad \mathcal{R}_2 = \begin{cases} \mathbf{g}(\mathbf{b}(\mathbf{x})) & \rightarrow \mathbf{g}(\mathbf{g}(\mathbf{x})) \\ \mathbf{g}(\mathbf{a}(\mathbf{x})) & \rightarrow \mathbf{b}(\mathbf{b}(\mathbf{g}(\mathbf{x}))) \end{cases}$$

with $\mathcal{C}_1 = \mathcal{C}_2 = \{\mathbf{a}, \mathbf{b}\}$, $\mathcal{F}_1 = \{\mathbf{f}\}$ and $\mathcal{F}_2 = \{\mathbf{g}\}$.

$\langle \mathcal{C}_1 \uplus \mathcal{F}_1, \mathcal{R}_1 \rangle$ admits the polynomial additive quasi-interpretation $[-]_{\mathbb{K}}^1$ defined by $[\mathbf{f}]_{\mathbb{K}}^1(X) = [\mathbf{a}]_{\mathbb{K}}^1(X) = X + 1$ and $[\mathbf{b}]_{\mathbb{K}}^1(X) = X + 2$.

$\langle \mathcal{C}_2 \uplus \mathcal{F}_2, \mathcal{R}_2 \rangle$ admits the polynomial additive quasi-interpretation $[-]_{\mathbb{K}}^2$ defined by $[\mathbf{g}]_{\mathbb{K}}^2(X) = [\mathbf{b}]_{\mathbb{K}}^2(X) = X + 1$ and $[\mathbf{a}]_{\mathbb{K}}^2(X) = X + 2$.

However their constructor-sharing union $\langle \mathcal{C}_1 \uplus \mathcal{F}_1, \mathcal{R}_1 \rangle \sqcup \langle \mathcal{C}_2 \uplus \mathcal{F}_2, \mathcal{R}_2 \rangle$ has no additive quasi-interpretation. Indeed suppose that such a quasi-interpretation $[-]_{\mathbb{K}}$ exists. $[\mathbf{f}]_{\mathbb{K}}$ and $[\mathbf{g}]_{\mathbb{K}}$ cannot be greater than a linear function because of the first rules of \mathcal{R}_1 and \mathcal{R}_2 . The second rule of \mathcal{R}_1 enforces that $[\mathbf{b}]_{\mathbb{K}}(X) \geq [\mathbf{a}]_{\mathbb{K}}([\mathbf{a}]_{\mathbb{K}}(X))$, whereas the second rule of \mathcal{R}_2 , enforces that $[\mathbf{a}]_{\mathbb{K}}(X) \geq [\mathbf{b}]_{\mathbb{K}}([\mathbf{b}]_{\mathbb{K}}(X))$. These constraints are not satisfiable by any additive quasi-interpretation.

The example below illustrates the non-modularity hierarchical union for additive polynomial quasi-interpretations.

Example 2.3.10. Consider the TRSs $\langle \mathcal{C}_1 \uplus \mathcal{F}_1, \mathcal{R}_1 \rangle$ and $\langle \mathcal{C}_2 \uplus \mathcal{F}_2, \mathcal{R}_2 \rangle$ defined by:

$$\mathcal{R}_1 = \begin{cases} \text{double}(0) & \rightarrow 0 \\ \text{double}(x+1) & \rightarrow \text{double}(x)+2 \end{cases} \quad \mathcal{R}_2 = \begin{cases} \text{exp}(0) & \rightarrow \underline{1} \\ \text{exp}(x+1) & \rightarrow \text{double}(\text{exp}(x)) \end{cases}$$

with $\mathcal{C}_1 = \{0, +1\}$, $\mathcal{C}_2 = \{0, +1, \text{double}\}$, $\mathcal{F}_1 = \{\text{double}\}$, and $\mathcal{F}_2 = \{\text{exp}\}$. $\langle \mathcal{C}_1 \uplus \mathcal{F}_1, \mathcal{R}_1 \rangle$ admits the additive polynomial quasi-interpretation $[-]_{\mathbb{K}}^1$ defined by $[\text{double}]_{\mathbb{K}}^1(X) = 2X$, $[+1]_{\mathbb{K}}^1(X) = X + 1$, and $[0]_{\mathbb{K}}^1 = 0$. $\langle \mathcal{C}_2 \uplus \mathcal{F}_2, \mathcal{R}_2 \rangle$ admits the additive polynomial quasi-interpretation $[-]_{\mathbb{K}}^2$ defined by $[\text{exp}]_{\mathbb{K}}^2(X) = X$, $[\text{double}]_{\mathbb{K}}^2(X) = [+1]_{\mathbb{K}}^2(X) = X + 1$, and $[0]_{\mathbb{K}}^2 = 0$. However their hierarchical union $\langle \mathcal{C}_1 \uplus \mathcal{F}_1, \mathcal{R}_1 \rangle \sqsubset \langle \mathcal{C}_2 \uplus \mathcal{F}_2, \mathcal{R}_2 \rangle$ has no additive polynomial quasi-interpretation.

The property of having an additive polynomial interpretation is not modular for constructor-sharing unions but [BMP07] has made the observation that the program obtained by such an union is still computing a polynomial time function. This can be combined with the fact that RPO is modular for constructor-sharing union. Let $\sqcup QI_{add}^{poly}(\mathbb{K}, \geq_{\mathbb{K}})$ be the set of function symbols whose TRS is the constructor-sharing union of two programs admitting an additive polynomial quasi-interpretation over \mathbb{K} .

Theorem 2.3.6 ([BMP07]). For $\mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+, \mathbb{R}^+\}$, we have:

- $[\sqcup QI_{add}^{poly}(\mathbb{K}, \geq_{\mathbb{K}}) \cap RPO\{p\}] = \text{FP}$,
- $[\sqcup QI_{add}^{poly}(\mathbb{K}, \geq_{\mathbb{K}}) \cap RPO\{l\}] = \text{FPSPACE}$.

Moreover, these characterizations are strictly more expressive.

Theorem 2.3.7. For $\mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+, \mathbb{R}^+\}$, $QI_{add}^{poly}(\mathbb{K}, \geq_{\mathbb{K}}) \subsetneq \sqcup QI_{add}^{poly}(\mathbb{K}, \geq_{\mathbb{K}})$.

Indeed, each TRS can be seen as the constructor sharing union of itself and a program with an empty set of rules. Example 2.3.9 illustrates that the inclusion is strict. Some more examples are provided in [BMP07]. However Theorem 2.3.6 does not hold for hierarchical unions as illustrated by Example 2.3.10 where both TRSs admit an additive polynomial interpretation but their hierarchical union computes an exponential function. The result can be recovered on hierarchical unions by putting some more restrictions. Given a polynomial P and a monomial m , we write $m \in P$ if $P = \alpha m + Q$, for some $\alpha \in \mathbb{N} - \{0\}$ and some polynomial Q such that $m \notin Q$. α is the coefficient of m in P , noted $\text{coeff}(m, P)$.

Definition 2.3.5. Given the hierarchical union of two TRSs $\langle \mathcal{C}_1 \uplus \mathcal{F}_1, \mathcal{R}_1 \rangle$ and $\langle \mathcal{C}_2 \uplus \mathcal{F}_2, \mathcal{R}_2 \rangle$ having the respective polynomial quasi-interpretations $[-]_{\mathbb{N}}^1$ and $[-]_{\mathbb{N}}^2$, $[-]_{\mathbb{N}}^1$ and $[-]_{\mathbb{N}}^2$ are kind preserving if for any symbol $\mathbf{b} \in \mathcal{F}_1 \cap \mathcal{C}_2$, the following conditions both hold:

- for any monomial m , $m \in [\mathbf{b}]_{\mathbb{N}}^1$ if and only if $m \in [\mathbf{b}]_{\mathbb{N}}^2$,
- for any monomial m , $\text{coeff}(m, [\mathbf{b}]_{\mathbb{N}}^1) = 1$ if and only if $\text{coeff}(m, [\mathbf{b}]_{\mathbb{N}}^2) = 1$.

Notice that the notion of kind comes from [BCMT98].

Example 2.3.11. The interpretations of the two TRSs of Example 2.3.10 are not kind preserving. Indeed $[\text{double}]_{\mathbb{K}}^1(X) = 2X$ and $[\text{double}]_{\mathbb{K}}^2(X) = X + 1$. Consequently, $\text{coeff}(X, [\mathbf{b}]_{\mathbb{N}}^1) = 2$ and $\text{coeff}(X, [\mathbf{b}]_{\mathbb{N}}^2) = 1$.

For a given TRS $\langle \mathcal{C} \uplus \mathcal{F}, \mathcal{R} \rangle$, we define an equivalence on function symbols $\mathbf{f} \approx_{\mathcal{F}} \mathbf{g}$ by the reflexive and transitive closure of whether \mathbf{f} called \mathbf{g} in \mathcal{R} and vice versa.

Definition 2.3.6. *The hierarchical union of two programs $\langle \mathcal{C}_1 \uplus \mathcal{F}_1, \mathcal{R}_1 \rangle$ and $\langle \mathcal{C}_2 \uplus \mathcal{F}_2, \mathcal{R}_2 \rangle$ is a stratified union, noted $\langle \mathcal{C}_1 \uplus \mathcal{F}_1, \mathcal{R}_1 \rangle \ll \langle \mathcal{C}_2 \uplus \mathcal{F}_2, \mathcal{R}_2 \rangle$ if the following conditions hold:*

- for any rule $\mathbf{f}(p_1, \dots, p_n) \rightarrow r \in \mathcal{R}_2$ and any subterm $\mathbf{g}(e_1, \dots, e_n)$ of r , if $\mathbf{f} \approx_{\mathcal{F}_2} \mathbf{g}$ then no shared function symbols of $\mathcal{C}_2 \cap \mathcal{F}_1$ occurs in e_1, \dots, e_n ,
- for any rule $\mathbf{f}(p_1, \dots, p_n) \rightarrow r \in \mathcal{R}_2$ there is no nesting of function symbols in r .

Let $\ll K PQI_{add}^{poly}(\mathbb{N}, \geq_{\mathbb{N}})$ be the set of function symbols whose TRS is the stratified union of two programs admitting an additive polynomial kind preserving quasi-interpretation over \mathbb{N} .²²

Theorem 2.3.8 ([BMP07]). *We have:*

- $\ll K PQI_{add}^{poly}(\mathbb{N}, \geq_{\mathbb{N}}) \cap RPO^{\{p\}} = \text{FP}$,
- $\ll K PQI_{add}^{poly}(\mathbb{N}, \geq_{\mathbb{N}}) \cap RPO^{\{l\}} = \text{FPSPACE}$.

An extension to \mathbb{Q}^+ and \mathbb{R}^+ is considered in [BMP07]. Basically, it just asks for some extra condition on the notion of kind preserving interpretation: all the multiplicative coefficients of the monomials have to be greater than 1. This is always true over \mathbb{N} .

Example 2.3.12. *Consider the TRSs $\langle \mathcal{C}_1 \uplus \mathcal{F}_1, \mathcal{R}_1 \rangle$ and $\langle \mathcal{C}_2 \uplus \mathcal{F}_2, \mathcal{R}_2 \rangle$ defined by:*

$$\mathcal{R}_1 = \begin{cases} \text{double}(0) & \rightarrow 0 \\ \text{double}(x+1) & \rightarrow \text{double}(x)+2 \\ \text{add}(0, y) & \rightarrow y \\ \text{add}(x+1, y) & \rightarrow \text{add}(x, y)+1 \end{cases} \quad \mathcal{R}_2 = \begin{cases} \text{sq}(0) & \rightarrow 0 \\ \text{sq}(x+1) & \rightarrow \text{add}(\text{sq}(x), \text{double}(x)) \end{cases}$$

with $\mathcal{C}_1 = \{0, +1\}$, $\mathcal{F}_1 = \{\text{double}, \text{add}\}$, $\mathcal{C}_2 = \{0, +1, \text{add}, \text{double}\}$, and $\mathcal{F}_2 = \{\text{sq}\}$. The function symbol sq computes the square of a unary number given as input in the hierarchical union $\langle \mathcal{C}_1 \uplus \mathcal{F}_1, \mathcal{R}_1 \rangle \sqsubset \langle \mathcal{C}_2 \uplus \mathcal{F}_2, \mathcal{R}_2 \rangle$. Neither $\text{sq} \approx_{\mathcal{F}} \text{double}$, nor $\text{sq} \approx_{\mathcal{F}} \text{add}$ hold. The program $\langle \mathcal{C}_2 \uplus \mathcal{F}_2, \mathcal{R}_2 \rangle$ is flat as there is no composition of function symbols in its rules. Moreover, there is no shared argument in the recursive call $\text{sq}(x)$. Consequently, $\langle \mathcal{C}_1 \uplus \mathcal{F}_1, \mathcal{R}_1 \rangle \ll \langle \mathcal{C}_2 \uplus \mathcal{F}_2, \mathcal{R}_2 \rangle$ is a stratified union.

Define the following quasi-interpretations $[-]_{\mathbb{N}}^1$ and $[-]_{\mathbb{N}}^2$ by:

$$\begin{array}{ll} [0]_{\mathbb{N}}^1 = 0 & [0]_{\mathbb{N}}^2 = 0, \\ [+1]_{\mathbb{N}}^1(X) = X + 1, & [+1]_{\mathbb{N}}^2(X) = X + 1, \\ [\text{add}]_{\mathbb{N}}^1(X, Y) = X + Y, & [\text{add}]_{\mathbb{N}}^2(X, Y) = X + Y + 1, \\ [\text{double}]_{\mathbb{N}}^1(X) = 3X, & [\text{double}]_{\mathbb{N}}^2(X) = 2X, \\ & [\text{sq}]_{\mathbb{N}}^2(X) = 2X^2. \end{array}$$

$[-]_{\mathbb{N}}^1$ and $[-]_{\mathbb{N}}^2$ are additive and polynomial kind preserving quasi-interpretations. The two programs can be ordered by RPO with product status and, consequently, the function symbols of the stratified union computes functions in FP.

²²Here the notion of additivity is slightly modified as we do not require symbols in $\mathcal{F}_1 \cap \mathcal{C}_2$ to be additive.

As a corollary of Theorem 2.3.7, we also have more expressive power for stratified union (constructor-sharing union being a particular case of stratified union):

Corollary 2.3.1. $QI_{add}^{poly}(\mathbb{N}, \geq_{\mathbb{N}}) \subsetneq \ll KPQI_{add}^{poly}(\mathbb{N}, \geq_{\mathbb{N}})$.

Due to the restrictions to kind preserving quasi-interpretations, stratified union does not improve greatly the expressive power of the method compared to constructor-sharing union. However they allow to improve quasi-interpretation synthesis by allowing to apply a divide-and-conquer strategy on TRS rules during inference.

2.4 Sup-interpretation

2.4.1 Motivations, definition, and basic properties

The subterm property drastically limits the quasi-interpretation domain. For example, a function defined by $\mathbf{f}(x, y) \rightarrow x$ has a quasi-interpretation at least equal to $[\mathbf{f}]_{\mathbb{K}}(X, Y) = \max_{\mathbb{K}}(X, Y)$, $\max_{\mathbb{K}}$ being the max function over \mathbb{K} , whereas one would expect it to be $[\mathbf{f}]_{\mathbb{K}}(X, Y) = X$, since the second parameter is dropped.

The notion of sup-interpretation (SI) was introduced in [MP06, MP09] in order to increase the intensionality of interpretation methods by compensating for such a drawback: sup-interpretations do not need to satisfy the subterm property.

As quasi-interpretations, sup-interpretations do not ensure termination and, consequently, they can be defined either on well-founded or on non well-founded ordered sets $\mathbb{K} \in \{\mathbb{N}, \mathbb{R}^+, \mathbb{Q}^+\}$.

Definition 2.4.1 (Sup-interpretation). *Given a TRS $\langle \mathcal{C} \uplus \mathcal{F}, \mathcal{R} \rangle$, a monotonic (additive and polynomial) assignment θ over \mathbb{K} is a (additive and polynomial) sup-interpretation over \mathbb{K} if for any $\mathbf{f} \in \mathcal{F}$ of arity m and for all $v_1, \dots, v_m \in \mathcal{T}(\mathcal{C})$:*

$$\mathbf{f}(v_1, \dots, v_m) \downarrow \implies [\mathbf{f}(v_1, \dots, v_m)]_{\mathbb{K}} \geq_{\mathbb{K}} [[\mathbf{f}]](v_1, \dots, v_m)_{\mathbb{K}},$$

where the sup-interpretation $[-]_{\mathbb{K}}$ is extended canonically to terms in a standard way.

As sup-interpretations have no subterm requirements, they allow us to consider non-subterm functions in a tighter way.

Example 2.4.1. *Consider the TRS computing the half of a unary number:*

$$\begin{aligned} \mathbf{half}(0) &\rightarrow 0, \\ \mathbf{half}(1) &\rightarrow 0, \\ \mathbf{half}(x+2) &\rightarrow \mathbf{half}(x)+1. \end{aligned}$$

It admits the following additive and polynomial sup-interpretation over \mathbb{Q}^+ :

$$\begin{aligned} [0]_{\mathbb{Q}^+} &= 0, \\ [+1]_{\mathbb{Q}^+}(X) &= X + 1, \\ [\mathbf{half}]_{\mathbb{Q}^+}(x) &= X/2. \end{aligned}$$

Indeed, $[[\mathbf{half}]](\underline{n}) = \underline{n}/2$, provided that $/2$ is the division over natural numbers and that \underline{n} is the unary encoding of the natural number n . Consequently, we check that for all \underline{n} ,

$$\begin{aligned} [\mathbf{half}(\underline{n})]_{\mathbb{Q}^+} &= [\mathbf{half}]_{\mathbb{Q}^+}([\underline{n}]_{\mathbb{Q}^+}) \\ &= [\underline{n}]_{\mathbb{Q}^+}/2 \\ &\geq_{\mathbb{Q}^+} [\underline{n}/2]_{\mathbb{Q}^+} = [[\mathbf{half}]](\underline{n})_{\mathbb{Q}^+}. \end{aligned}$$

However finding a sup-interpretation of all the function symbols of a given TRS consists in finding a (partial) upper-bound on all their computation and is not feasible. It is shown in [Péc13] that the sup-interpretation verification problem is Π_1^0 -complete and that the synthesis problem is in Σ_3^0 . Some criteria were developed to overcome this issue.

In [MP09], a criterion was developed to ensure polynomial space computations using sup-interpretations. For that purpose, let $\succeq_{\mathcal{F}}$ be an ordering on function symbols \mathcal{F} of a given TRS defined by $\mathbf{f} \succeq_{\mathcal{F}} \mathbf{g}$ if there are a context $C[\diamond]$ and terms $p_1, \dots, p_n, t_1, \dots, t_m$ such that $\mathbf{f}(p_1, \dots, p_n) \rightarrow C[\mathbf{g}(t_1, \dots, t_m)] \in \mathcal{R}$. $\mathbf{f} \approx_{\mathcal{F}} \mathbf{g}$ if both $\mathbf{f} \succeq_{\mathcal{F}} \mathbf{g}$ and $\mathbf{g} \succeq_{\mathcal{F}} \mathbf{f}$ hold. $\mathbf{f} \succ_{\mathcal{F}} \mathbf{g}$ if $\mathbf{f} \succeq_{\mathcal{F}} \mathbf{g}$ holds and $\mathbf{g} \succeq_{\mathcal{F}} \mathbf{f}$ does not hold.

Definition 2.4.2. *In a TRS a rule $\mathbf{f}(p_1, \dots, p_n) \rightarrow C[\mathbf{g}_1(\bar{e}_1), \dots, \mathbf{g}_m(\bar{e}_m)]$ is a fraternity if:*

- $\forall i \leq m, \mathbf{g}_i \approx_{\mathcal{F}} \mathbf{f}$,
- $\forall \mathbf{h} \in C[\diamond_1, \dots, \diamond_m], \mathbf{f} \succ_{\mathcal{F}} \mathbf{h}$.

Definition 2.4.3. *For $\mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+, \mathbb{R}^+\}$, a TRS is quasi-friendly (in $QF_{\mathbb{K}}$) if there is a polynomial and additive sup-interpretation $[-]_{\mathbb{K}}$ over \mathbb{K} and a (partial) polynomial, monotonic, and subterm assignment ω over \mathbb{K} , called weight, such that for each fraternity $\mathbf{f}(p_1, \dots, p_n) \rightarrow C[\mathbf{g}_1(\bar{e}_1), \dots, \mathbf{g}_m(\bar{e}_m)]$ the following holds:*

$$\omega(\mathbf{f})([p_1]_{\mathbb{K}}, \dots, [p_n]_{\mathbb{K}}) \geq_{\mathbb{K}} [C]_{\mathbb{K}}(\omega(\mathbf{g}_1)([\bar{e}_1]_{\mathbb{K}}), \dots, \omega(\mathbf{g}_m)([\bar{e}_m]_{\mathbb{K}})),$$

where $[C]_{\mathbb{K}}$ is the function $([\diamond_1]_{\mathbb{K}}, \dots, [\diamond_m]_{\mathbb{K}}) \mapsto [C[\diamond_1, \dots, \diamond_m]]_{\mathbb{K}}$ and where the interpretation of a sequence $\bar{t} = t_1, \dots, t_k$ is defined by $[\bar{t}]_{\mathbb{K}} = [t_1]_{\mathbb{K}}, \dots, [t_k]_{\mathbb{K}}$.

We have a first result on the space consumption of a TRS in $QF_{\mathbb{K}}$.

Proposition 2.4.1. *Given a TRS $\langle \mathcal{C} \uplus \mathcal{F}, \mathcal{R} \rangle$ in $QF_{\mathbb{K}}$, there is a polynomial P over \mathbb{K} such that for any $\mathbf{f} \in \mathcal{F}$ of arity n and any values $v_1, \dots, v_n \in \mathcal{T}(\mathcal{C})$, if $\mathbf{f}(v_1, \dots, v_n) \downarrow$ then $|\llbracket \mathbf{f} \rrbracket(v_1, \dots, v_n)| \leq_{\mathbb{K}} P(|v_1|, \dots, |v_n|)$.*

Example 2.4.2. *The TRS of Example 2.4.1 has only one fraternity*

$$\mathbf{half}(x+2) \rightarrow \mathbf{half}(x)+1$$

as \mathbf{half} is the only function symbol and $\approx_{\mathcal{F}}$ is reflexive. Now $[+1]_{\mathbb{K}}(X) = X + 1$ defines an additive and polynomial sup-interpretation. For the TRS to be in $QF_{\mathbb{Q}^+}$, one has to find a weight (subterm, monotonic, and polynomial assignment) ω such that

$$\omega(\mathbf{half})([x+2]_{\mathbb{K}}) \geq_{\mathbb{Q}^+} [+1]_{\mathbb{K}}(\omega(\mathbf{half})([x]_{\mathbb{K}}))$$

or equivalently

$$\omega(\mathbf{half})(X + 2) \geq_{\mathbb{Q}^+} \omega(\mathbf{half})(X) + 1.$$

It suffices to set $\omega(\mathbf{half})(X) = X$ and we can conclude that the TRS is in $QF_{\mathbb{Q}^+}$.

Again, the notion of sup-interpretation has to be combined with some termination argument in order to characterize complexity classes.

Theorem 2.4.1 ([MP09]). *For $\mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+, \mathbb{R}^+\}$, the following characterizations hold:*

- $\llbracket QF_{\mathbb{K}} \cap RPO^{\{p\}} \rrbracket = \text{FP}$,

- $\llbracket QF_{\mathbb{K}} \cap RPO^{\{l\}} \rrbracket = \text{FPSPACE}$.

Example 2.4.3. The TRS of Example 2.4.1 is in $QF_{\mathbb{K}}$ and can trivially be oriented by \prec_{rpo} . Consequently, the computed function $\llbracket \text{half} \rrbracket$ is in FP.

It is worth noticing that the quasi-friendly criterion has also been extended in [MP09] to:

- non-terminating programs over stream data in a criterion called quasi-friendly with bounded recursive calls,
- divide-and-conquer algorithms and, in particular `quicksort`, in a criterion called quasi-friendly modulo projection.

Moreover, combinations with the termination methods such as DPs and SCP have also been considered. We will focus on an adaptation to DPs, a complete method for showing program termination, in the next subsections.

2.4.2 Combination with the dependency pair method

RPO termination techniques have an inherent syntactic restriction on the shape of admissible recursions to avoid the computation of non-primitive recursive functions. To overcome these (intensionality) issues (the use of RPO and the subterm property requirement), the notion of dependency pair (DP), introduced by Arts and Giesl [AG00], was combined with the notion of interpretation in [MP08c, MP09] in order to characterize FP and FPSPACE.

DPs provides a method for showing program termination that is complete with respect to termination and, consequently, captures strictly more programs than RPO, as RPO was shown to be NP-complete in [KN85] and, hence, cannot be complete for termination.

We start by briefly reviewing the DP method.

Definition 2.4.4 (Dependency Pair). *Given a TRS $\langle \mathcal{C} \uplus \mathcal{F}, \mathcal{R} \rangle$, the set of dependency pair symbols $\mathcal{F}^{\#}$ is defined by $\mathcal{F}^{\#} = \mathcal{F} \cup \{\mathbf{f}^{\#} \mid \mathbf{f} \in \mathcal{F}\}$, $\mathbf{f}^{\#}$ being a fresh function symbol of the same arity as \mathbf{f} . Given a term $t = \mathbf{f}(t_1, \dots, t_n)$, let $t^{\#}$ be a notation for $\mathbf{f}^{\#}(t_1, \dots, t_n)$. A dependency pair is a pair $l^{\#} \rightarrow u^{\#}$ if $u^{\#} = \mathbf{g}^{\#}(t_1, \dots, t_n)$, for some $\mathbf{g} \in \mathcal{F}$, and if there is a context $C[\diamond]$ such that $l \rightarrow C[u] \in \mathcal{R}$ and u is not a proper subterm of l . Let $DP(\mathcal{R})$ be the set of all dependency pairs in $\langle \mathcal{C} \uplus \mathcal{F}, \mathcal{R} \rangle$.*

Definition 2.4.5 (Dependency Pair Graph). *The dependency pair graph (DPG) of a given TRS is the graph obtained as follows:*

- the vertices are the dependency pairs,
- there is an edge from $t \rightarrow s$ to $t' \rightarrow s'$, if there is a substitution σ such that $s\sigma \rightarrow_{\mathcal{R}}^* t'\sigma$.

provided that \mathcal{R}' is the program obtained by extending \mathcal{R} with a new rule for each dependency pair.

A cycle of the DPG is a cycle in the corresponding graph structure.

Definition 2.4.6. A reduction pair is a couple $(\geq, >)$ such that:

- \geq is a (weakly) monotonic quasi-ordering over terms stable by substitution,
- $>$ is a well-founded ordering over terms and stable by substitution,

satisfying $\geq \circ > \subseteq >$ or $> \circ \geq \subseteq >$. If both conditions are satisfied then $(\geq, >)$ is called a strong reduction pair.

Example 2.4.4. $(\geq_{\mathbb{N}}, >_{\mathbb{N}})$ and $(\geq_{\mathbb{R}^+}, >_{\mathbb{R}^+}^{wf})$ are strong reduction pairs but $(\geq_{\mathbb{Q}^+}, >_{\mathbb{Q}^+})$ is neither a strong reduction pair, nor a reduction pair, as $>_{\mathbb{Q}^+}$ is not well-founded.

Definition 2.4.7. Given a reduction pair $(\geq, >)$, let $DP(\geq, >)$ be the set of TRSs such that:

- for each rule $l \rightarrow r$, $l \geq r$,
- for each DP $s \rightarrow t$ in a cycle of the DPG, $s \geq t$,
- for each cycle of the DPG, there is a dependency pair $s \rightarrow t$ such that $s > t$.

In the particular case where $>$ is the strict order of \geq , we simply write $DP(\geq)$.

The complete characterization of termination can be stated as follows.

Theorem 2.4.2 ([AG00]). A TRS $\langle \mathcal{C} \uplus \mathcal{F}, \mathcal{R} \rangle$ is terminating if and only if there is a well-founded monotonic quasi-ordering \geq closed under substitution such that $\langle \mathcal{C} \uplus \mathcal{F}, \mathcal{R} \rangle \in DP(\geq)$.

The implication from right to left is easy to grasp as any infinite reduction sequence will correspond to an infinite descending chain with respect to the quasi-ordering \geq . Infinitely many strict decreases have to occur in such a chain and correspond to a path involving infinitely many cycles in the DPG. This contradicts the well-foundedness assumption on the strict order corresponding to \geq .

As termination is an undecidable property, it is undecidable to show that a TRS satisfy the DP requirements for some reduction pair. The undecidability of this characterization is hidden behind the difficulty of generating the DPG of a given TRS. Indeed, deciding whether there is a connecting edge between two dependency pairs is an undecidable property. Some simplifications to generate the DPG are considered in [AG00].

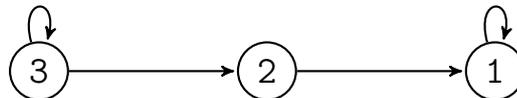
Example 2.4.5. Consider the following TRS computing the logarithm over unary numbers:

$$\begin{aligned} \text{half}(0) &\rightarrow 0, \\ \text{half}(1) &\rightarrow 0, \\ \text{half}(x+2) &\rightarrow \text{half}(x)+1, \\ \log(x+2) &\rightarrow \log(\text{half}(x+2))+1, \\ \log(1) &\rightarrow 0. \end{aligned}$$

It admits the following DPs:

$$\begin{aligned} 1 &: \text{half}^\sharp(x+2) \rightarrow \text{half}^\sharp(x), \\ 2 &: \log^\sharp(x+2) \rightarrow \text{half}^\sharp(x+2), \\ 3 &: \log^\sharp(x+2) \rightarrow \log^\sharp(\text{half}(x+2)), \end{aligned}$$

and has the following DPG:



Consequently, showing termination just consists in finding a well-founded quasi-ordering \geq such that for all rule $l \rightarrow r$, $l \geq r$ and the following inequalities hold:

$$\begin{aligned} \mathbf{half}^\sharp(\mathbf{x}+2) &> \mathbf{half}^\sharp(\mathbf{x}), \\ \mathbf{log}^\sharp(\mathbf{x}+2) &\geq \mathbf{half}^\sharp(\mathbf{x}+2), \\ \mathbf{log}^\sharp(\mathbf{x}+2) &> \mathbf{log}^\sharp(\mathbf{half}(\mathbf{x}+2)). \end{aligned}$$

The two strict inequalities above correspond to the two cycles in the DPG.

In [MP09], an alternative characterization of FPSPACE was provided using the DP method.

Definition 2.4.8. For $\mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+, \mathbb{R}^+\}$, a program has strictly bounded recursive calls (is in $SBR_{\mathbb{K}}$) if there are a polynomial and additive sup-interpretation $[-]_{\mathbb{K}}$ over \mathbb{K} and a polynomial and subterm weight ω over \mathbb{K} such that:

- it is in $QF_{\mathbb{K}}$ with respect to $[-]_{\mathbb{K}}$ and ω ,
i.e. for each fraternity $\mathbf{f}(p_1, \dots, p_n) \rightarrow C[\mathbf{g}_1(\bar{e}_1), \dots, \mathbf{g}_m(\bar{e}_m)]$ the following holds:

$$\omega(\mathbf{f})([p_1]_{\mathbb{K}}, \dots, [p_n]_{\mathbb{K}}) \geq_{\mathbb{K}} [C]_{\mathbb{K}}(\omega(\mathbf{g}_1)([\bar{e}_1]_{\mathbb{K}}), \dots, \omega(\mathbf{g}_m)([\bar{e}_m]_{\mathbb{K}})),$$

- for each DP $\mathbf{f}^\sharp(\bar{p}) \rightarrow \mathbf{g}^\sharp(\bar{e})$, we have:

$$\omega(\mathbf{f})([\bar{p}]_{\mathbb{K}}) \geq_{\mathbb{K}} \omega(\mathbf{g})([\bar{e}]_{\mathbb{K}}),$$

- for each cycle in the DPG, there is at least one DP $\mathbf{f}^\sharp(\bar{p}) \rightarrow \mathbf{g}^\sharp(\bar{e})$ such that:

$$\omega(\mathbf{f})([\bar{p}]_{\mathbb{K}}) >_{\mathbb{K}}^{wf} \omega(\mathbf{g})([\bar{e}]_{\mathbb{K}}).$$

Example 2.4.6. The TRS of Example 2.4.5 has already been shown to be in $QF_{\mathbb{Q}^+}$ in Example 2.4.2 for the weight ω defined by $\omega(\mathbf{log})(X) = \omega(\mathbf{half})(X) = X$. Consequently, for this TRS belongs to $SBR_{\mathbb{Q}^+}$, it remains to show:

$$\begin{aligned} \omega(\mathbf{half})([\mathbf{x}+2]_{\mathbb{Q}^+}) &>_{\mathbb{Q}^+}^{wf} \omega(\mathbf{half})([\mathbf{x}]_{\mathbb{Q}^+}), \\ \omega(\mathbf{log})([\mathbf{x}+2]_{\mathbb{Q}^+}) &\geq_{\mathbb{Q}^+} \omega(\mathbf{half})([\mathbf{x}+2]_{\mathbb{Q}^+}), \\ \omega(\mathbf{log})([\mathbf{x}+2]_{\mathbb{Q}^+}) &>_{\mathbb{Q}^+}^{wf} \omega(\mathbf{log})([\mathbf{half}(\mathbf{x}+2)]_{\mathbb{Q}^+}). \end{aligned}$$

It was shown in Example 2.4.1, that the assignment $[-]_{\mathbb{K}}$ defined by $[+1]_{\mathbb{Q}^+}(X) = X + 1$ and $[\mathbf{half}]_{\mathbb{Q}^+}(X) = X/2$ is a suitable sup-interpretation for \mathbf{half} . The above inequalities can be reformulated as follows:

$$\begin{aligned} \omega(\mathbf{half})(X + 2) &>_{\mathbb{Q}^+}^{wf} \omega(\mathbf{half})(X), \\ \omega(\mathbf{log})(X + 2) &\geq_{\mathbb{Q}^+} \omega(\mathbf{half})(X + 2), \\ \omega(\mathbf{log})(X + 2) &>_{\mathbb{Q}^+}^{wf} \omega(\mathbf{log})(X/2 + 1). \end{aligned}$$

They are clearly satisfied for the weight defined above together with the well-founded ordering $>_1$. Consequently, both $[\mathbf{log}]$ and $[\mathbf{half}]$ are in FPSPACE (we will improve this result soon).

Theorem 2.4.3 ([MP09]). For $\mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+, \mathbb{R}^+\}$, $[[SBR_{\mathbb{K}}]] = \text{FPSPACE}$.

Example 2.4.7. Consider the following program taken from [BMM11] and computing the QBF problem:

$$\begin{aligned}
& \text{not}(\text{tt}) \rightarrow \text{ff}, \\
& \text{not}(\text{ff}) \rightarrow \text{tt}, \\
& \text{or}(\text{tt}, x) \rightarrow \text{tt}, \\
& \text{or}(\text{ff}, x) \rightarrow x, \\
& 0 = 0 \rightarrow \text{tt}, \\
& \text{suc}(x) = 0 \rightarrow \text{ff}, \\
& 0 = \text{suc}(y) \rightarrow \text{ff}, \\
& \text{suc}(x) = \text{suc}(y) \rightarrow x = y, \\
& \text{in}(x, \text{nil}) \rightarrow \text{ff}, \\
& \text{in}(x, \text{cons}(a, l)) \rightarrow \text{or}(x = a, \text{in}(x, l)), \\
& \text{verify}(\text{Var}(x), t) \rightarrow \text{in}(x, t), \\
& \text{verify}(\text{Not}(u), t) \rightarrow \text{not}(\text{verify}(u, t)), \\
& \text{verify}(\text{Or}(u, v), t) \rightarrow \text{or}(\text{verify}(u, t), \text{verify}(v, t)), \\
& \text{verify}(\text{Exists}(n, u), t) \rightarrow \text{or}(\text{verify}(u, \text{cons}(n, t)), \text{verify}(u, t)), \\
& \text{qbf}(u) \rightarrow \text{verify}(u, \text{nil}).
\end{aligned}$$

The following precedence holds $\{\text{not}, \text{or}, =\} \prec_{\mathcal{F}} \text{in} \prec_{\mathcal{F}} \text{verify} \prec_{\mathcal{F}} \text{qbf}$. Consider the additive and polynomial sup-interpretation $[-]_{\mathbb{K}}$ defined by:

$$\begin{aligned}
& [\text{suc}]_{\mathbb{K}}(X) = X + 1, \\
& [\text{cons}]_{\mathbb{K}}(X, Y) = X + Y + 1, \\
& [\text{Or}]_{\mathbb{K}}(X, Y) = X + Y + 1, \\
& [\text{or}]_{\mathbb{K}}(X, Y) = 0, \\
& [\text{Not}]_{\mathbb{K}}(X) = X + 1, \\
& [\text{not}]_{\mathbb{K}}(X) = 0, \\
& [\text{Exists}]_{\mathbb{K}}(X, Y) = X + Y + 2,
\end{aligned}$$

and the subterm, monotonic, and polynomial weight ω defined by:

$$\begin{aligned}
& \omega(=)(X, Y) = X + Y, \\
& \omega(\text{in})(X, Y) = X + Y, \\
& \omega(\text{verify})(X, Y) = X + Y.
\end{aligned}$$

It is easy to check that the TRS is in $\text{SBR}_{\mathbb{K}}$ and, consequently, the computed function is in FPSPACE .

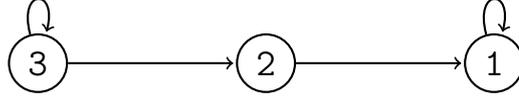
Example 2.4.8. As a counter-example, consider the TRS of Example 2.2.1:

$$\begin{aligned}
& \text{double}(0) \rightarrow 0, \\
& \text{double}(x+1) \rightarrow \text{double}(x)+2, \\
& \text{exp}(0) \rightarrow \underline{1}, \\
& \text{exp}(x+1) \rightarrow \text{double}(\text{exp}(x)).
\end{aligned}$$

It admits the following DPs:

$$\begin{aligned} 1 &: \text{double}^\sharp(\mathbf{x}+2) \rightarrow \text{double}^\sharp(\mathbf{x}), \\ 2 &: \text{exp}^\sharp(\mathbf{x}+1) \rightarrow \text{double}^\sharp(\text{exp}(\mathbf{x})), \\ 3 &: \text{exp}^\sharp(\mathbf{x}+1) \rightarrow \text{exp}^\sharp(\mathbf{x}), \end{aligned}$$

and has the following DPG:



For this TRS to be in $SBR_{\mathbb{K}}$, it has to be in $QF_{\mathbb{K}}$ and the following property has to be satisfied:

$$\omega(\text{exp})([\mathbf{x}+1]_{\mathbb{K}}) \geq [\text{double}]_{\mathbb{K}}(\omega(\text{exp}(\mathbf{x}))),$$

for some polynomial and additive sup-interpretation $[-]_{\mathbb{K}}$ and some monotonic, polynomial, and subterm weight ω . As $[\text{double}]_{\mathbb{K}}$ is a sup-interpretation, it has to satisfy $\forall n, [\text{double}]_{\mathbb{K}}(n) \geq [[\text{double}]]_{\mathbb{K}}(n) = 2n$. Consequently, the above inequality can be transformed into:

$$\omega(\text{exp})(X + 1) \geq [\text{double}]_{\mathbb{K}}(\omega(\text{exp})(X)) \geq 2\omega(\text{exp})(X).$$

Clearly, no polynomial $\omega(\text{exp})$ can satisfy this inequality.

An alternative and intensionally equivalent variant of the result of Theorem 2.4.3 was introduced in [Bon11] using non-subterm assignments rather than sup-interpretations: the symbols \mathbf{f}^\sharp are required to have a subterm assignments. This role is played by the weight $\omega(\mathbf{f})$ in the above setting. Moreover, all the rewrite rules are oriented in [Bon11] whereas this is not needed for sup-interpretations in Definition 2.4.8. The result can be rephrased as follows.

Definition 2.4.9. For a given assignment $[-]_{\mathbb{K}}$ of a TRS over $\mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+, \mathbb{R}^+\}$, define $\geq_{\mathbb{K}}^{[-]_{\mathbb{K}}}$ (and $>_{\mathbb{K}}^{[-]_{\mathbb{K}}}$, respectively) by:

$$\forall s, t \in \mathcal{T}(\Sigma, \mathcal{X}), s \geq_{\mathbb{K}}^{[-]_{\mathbb{K}}} t \text{ if and only if } [s]_{\mathbb{K}} \geq_{\mathbb{K}}^{wf} [t]_{\mathbb{K}},$$

(and $s >_{\mathbb{K}}^{[-]_{\mathbb{K}}} t$ if and only if $[s]_{\mathbb{K}} >_{\mathbb{K}}^{wf} [t]_{\mathbb{K}}$, respectively).

Notice that the set $DP(\geq_{\mathbb{K}}^{[-]_{\mathbb{K}}})$ is clearly defined as $>_{\mathbb{K}}^{[-]_{\mathbb{K}}}$ is well-founded, by definition.

Definition 2.4.10. Let \mathcal{A} be the set of assignments $[-]_{\mathbb{K}}$ that are additive, monotonic, and polynomial. Moreover, let \mathcal{AS} be the subset of assignments in \mathcal{A} such that for each symbol $\mathbf{f} \in \mathcal{F}$ $[\mathbf{f}^\sharp]_{\mathbb{K}}$ has the subterm property.

We are now ready to reformulate $SBR_{\mathbb{K}}$ using the notion of DP.

Theorem 2.4.4. For $\mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+, \mathbb{R}^+\}$, $SBR_{\mathbb{K}} = \cup_{[-]_{\mathbb{K}} \in \mathcal{AS}} \{DP(\geq_{\mathbb{K}}^{[-]_{\mathbb{K}}})\}$.

The proof is easy as it just consists in defining an assignment $[-]_{\mathbb{K}}'$ such that for each symbol $\mathbf{b} \in \mathcal{C} \uplus \mathcal{F}$, $[\mathbf{b}]_{\mathbb{K}}' := [\mathbf{b}]_{\mathbb{K}}$ and for each function symbol $\mathbf{f} \in \mathcal{F}$, $[\mathbf{f}^\sharp]_{\mathbb{K}}' := \omega(\mathbf{f})$.

Given a term l^\sharp let its neighborhood $N(l^\sharp)$ be defined by $t^\sharp \in N(l^\sharp)$ if and only if (l^\sharp, t^\sharp) is involved in at least one cycle of the DPG. Alternative characterizations of FP and FPSPACE were provided in [MP08c] as follows.

Definition 2.4.11. For a given assignment $[-]_{\mathbb{K}}$ over $\mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+, \mathbb{R}^+\}$, a TRS has:

- Bounded recursive calls (is in $BRC([-]_{\mathbb{K}})$), if for each DP $\mathbf{f}^{\sharp}(p_1, \dots, p_n) \rightarrow \mathbf{g}^{\sharp}(t_1, \dots, t_m)$ in a cycle of the DPG, we have:

$$\sum_{i=1}^n [p_i]_{\mathbb{K}} \geq_{\mathbb{K}} \sum_{j=1}^m [t_j]_{\mathbb{K}}.$$

- Bounded neighborhood (is in $BN([-]_{\mathbb{K}})$), if for each neighborhood $N(l^{\sharp}) = \{t_1^{\sharp}, \dots, t_n^{\sharp}\}$, we have:

$$[l]_{\mathbb{K}} >_{\mathbb{K}}^{wf} \sum_{i=1}^n [t_i]_{\mathbb{K}}.$$

For $\mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+, \mathbb{R}^+\}$, let $DP(TIME_{\mathbb{K}}) = \cup_{[-]_{\mathbb{K}} \in \mathcal{A}} \{DP(\geq_{\mathbb{K}}^{[-]_{\mathbb{K}}}) \cap BRC([-]_{\mathbb{K}}) \cap BN([-]_{\mathbb{K}})\}$ and $DP(SPACE_{\mathbb{K}}) = \cup_{[-]_{\mathbb{K}} \in \mathcal{A}} \{DP(\geq_{\mathbb{K}}^{[-]_{\mathbb{K}}}) \cap BRC([-]_{\mathbb{K}})\}$. We are now able to give the characterizations of polynomial time and polynomial space.

Theorem 2.4.5 ([MP08c]). For $\mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+, \mathbb{R}^+\}$, the following characterizations hold:

- $\llbracket DP(TIME_{\mathbb{K}}) \rrbracket = \text{FP}$,
- $\llbracket DP(SPACE_{\mathbb{K}}) \rrbracket = \text{FPSPACE}$.

Example 2.4.9. As a direct application of Theorem 2.4.4, we show that the TRS of Example 2.4.6 is in $DP(TIME_{\mathbb{Q}^+})$. Now we have two dependency pairs involved in cycles of the DPG:

$$\begin{aligned} \text{half}^{\sharp}(\mathbf{x}+2) &\rightarrow \text{half}^{\sharp}(\mathbf{x}), \\ \log^{\sharp}(\mathbf{x}+2) &\rightarrow \log^{\sharp}(\text{half}(\mathbf{x}+2)). \end{aligned}$$

Hence for this TRS to have bounded recursive calls, we need to check that:

$$\begin{aligned} [\mathbf{x} + 2]_{\mathbb{Q}^+} &\geq_{\mathbb{Q}^+} [\mathbf{x}]_{\mathbb{Q}^+}, \\ [\mathbf{x} + 2]_{\mathbb{Q}^+} &\geq_{\mathbb{Q}^+} [\text{half}(\mathbf{x}+2)]_{\mathbb{Q}^+}. \end{aligned}$$

These inequalities are satisfied for the additive and polynomial sup-interpretation defined by $[\text{half}]_{\mathbb{Q}^+}(X) = X/2$ and $[+1]_{\mathbb{Q}^+}(X) = X + 1$. Moreover we have $N(\text{half}^{\sharp}(\mathbf{x} + 2)) = \{\text{half}^{\sharp}(\mathbf{x})\}$ and $N(\log^{\sharp}(\mathbf{x}+2)) = \{\log^{\sharp}(\text{half}(\mathbf{x}+2))\}$. Hence for this TRS to have bounded neighborhoods, we need to check that:

$$\begin{aligned} [\text{half}(\mathbf{x}+2)]_{\mathbb{Q}^+} &>_{\mathbb{Q}^+}^{wf} [\text{half}(\mathbf{x})]_{\mathbb{Q}^+}, \\ [\log(\mathbf{x}+2)]_{\mathbb{Q}^+} &>_{\mathbb{Q}^+}^{wf} [\log(\text{half}(\mathbf{x}+2))]_{\mathbb{Q}^+}. \end{aligned}$$

These inequalities are satisfied for the above additive and polynomial sup-interpretation extended by $[\log]_{\mathbb{Q}^+}(X) = X$ with respect to the well-founded ordering $>_1$.

Example 2.4.10. The TRS of Example 2.4.7 is in $DP(SPACE_{\mathbb{K}})$. Indeed, it is in $SBR_{\mathbb{K}}$, for some sup-interpretation $[-]_{\mathbb{K}}$ and weight ω , and, consequently, in $DP(\geq_{\mathbb{K}}^{[-]_{\mathbb{K}}})$, for the assignment $[-]'_{\mathbb{K}}$ in \mathcal{AS} (and a fortiori in \mathcal{A}) defined by $[\mathbf{b}]'_{\mathbb{K}} := [\mathbf{b}]_{\mathbb{K}}, \forall \mathbf{b} \in \mathcal{C} \uplus \mathcal{F}$, and $[\mathbf{f}^{\sharp}]'_{\mathbb{K}} := \omega(\mathbf{f}), \forall \mathbf{f} \in \mathcal{F}$. It is also in $BRC([-]'_{\mathbb{K}})$ as the functions $\omega(\mathbf{b})$ provided in Example 2.4.7 are all linear. Consequently, $\omega(\mathbf{f})([\bar{p}]'_{\mathbb{K}}) \geq_{\mathbb{K}} \omega(\mathbf{g})([\bar{e}]'_{\mathbb{K}})$ can be rewritten into:

$$\sum_{i=1}^n [p_i]'_{\mathbb{K}} \geq_{\mathbb{K}} \sum_{j=1}^m [t_j]'_{\mathbb{K}}.$$

However it is (hopefully) not in $DP(TIME_{\mathbb{K}})$. Indeed, consider the rule

$$\text{verify}(\text{Exists}(\mathbf{n}, \mathbf{u}), \mathbf{t}) \rightarrow \text{or}(\text{verify}(\mathbf{u}, \text{cons}(\mathbf{n}, \mathbf{t})), \text{verify}(\mathbf{u}, \mathbf{t})),$$

the neighborhood $N(\text{verify}^{\#}(\text{Exists}(\mathbf{n}, \mathbf{u}), \mathbf{t}))$ is equal to $\{\text{verify}^{\#}(\mathbf{u}, \text{cons}(\mathbf{n}, \mathbf{t})), \text{verify}^{\#}(\mathbf{u}, \mathbf{t})\}$. Consequently, for the program to be in $SN([-]_{\mathbb{K}})$, one has to check that:

$$[\text{verify}(\text{Exists}(\mathbf{n}, \mathbf{u}), \mathbf{t})]_{\mathbb{K}}' >_{\mathbb{K}} [\text{verify}(\mathbf{u}, \text{cons}(\mathbf{n}, \mathbf{t}))]_{\mathbb{K}}' + [\text{verify}(\mathbf{u}, \mathbf{t})]_{\mathbb{K}}'$$

that can be restated as:

$$[\text{verify}]_{\mathbb{K}}'(N + U + 1, T) >_{\mathbb{K}} [\text{verify}]_{\mathbb{K}}'(U, N + T + 1) + [\text{verify}]_{\mathbb{K}}'(U, T) \geq 2[\text{verify}]_{\mathbb{K}}'(U, T),$$

which is clearly not satisfiable by any polynomial and monotonic assignment.

2.4.3 Sup-interpretation vs (quasi-)interpretation

Let $SI_{add}^{poly}(\mathbb{K}, \geq_{\mathbb{K}})$ be the set of function symbols whose TRS admits an additive and polynomial sup-interpretation over \mathbb{K} . We obtain an intensionality result, similar to the one of Theorem 2.3.1, as the set of programs admitting an additive and polynomial quasi-interpretations is strictly included in the set of programs admitting an additive and polynomial sup-interpretation.²³

Theorem 2.4.6 ([MP09]²⁴). For $\mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+, \mathbb{R}^+\}$, $QI_{add}^{poly}(\mathbb{K}, \geq_{\mathbb{K}}) \subsetneq SI_{add}^{poly}(\mathbb{K}, \geq_{\mathbb{K}})$.

The inclusion is obtained as a direct consequence of Lemma 2.3.1. The strict inclusion is highlighted by the following example.

Example 2.4.11. Consider the TRS of Example 2.4.5:

$$\begin{aligned} \text{half}(0) &\rightarrow 0, \\ \text{half}(1) &\rightarrow 0, \\ \text{half}(\mathbf{x}+2) &\rightarrow \text{half}(\mathbf{x})+1, \\ \log(\mathbf{x}+2) &\rightarrow \log(\text{half}(\mathbf{x}+2))+1, \\ \log(1) &\rightarrow 0. \end{aligned}$$

Notice that because of the fifth rule, the program cannot be oriented by RPO. Moreover, it does not admit an interpretation or a quasi-interpretation as, by subterm and monotonicity properties, it would imply the following contradiction:

$$[\log^{\#}]_{\mathbb{K}}(X + 2k) > [\log^{\#}]_{\mathbb{K}}([\text{half}]_{\mathbb{K}}(X + 2k)) \geq [\log^{\#}]_{\mathbb{K}}(X + 2k),$$

for $[+1]_{\mathbb{K}}(X) = k$.

However it admits the following additive and polynomial sup-interpretation over \mathbb{Q}^+ :

$$\begin{aligned} [0]_{\mathbb{K}} &= 0, \\ [+1]_{\mathbb{K}}(X) &= X + 1, \\ [\text{half}]_{\mathbb{K}}(X) &= X/2, \\ [\log]_{\mathbb{K}}(X) &= X. \end{aligned}$$

²³Again, this result remains true if additivity and polynomiality properties are withdrawn.

²⁴There is a small distinction with [MP09] where additivity is included in the definition of a sup-interpretation.

Combining Theorem 2.3.1 and Theorem 2.4.6, we obtain the following straightforward corollary.

Corollary 2.4.1. *For $\mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+, \mathbb{R}^+\}$, $I_{add}^{poly}(\mathbb{K}, \geq_{\mathbb{K}}^{wf}) \subsetneq SI_{add}^{poly}(\mathbb{K}, \geq_{\mathbb{K}})$.*

In order to obtain intensional results, it is of interest to compare: $QI_{add}^{poly}(\mathbb{K}, \geq_{\mathbb{K}})$ and $QF_{\mathbb{K}}$. As any quasi-interpretation is a sup-interpretation and as any quasi-interpretation is a weight (see Definition 2.4.3), we obtain the following result.

Theorem 2.4.7 ([MP09]). *For $\mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+, \mathbb{R}^+\}$, $QI_{add}^{poly}(\mathbb{K}, \geq_{\mathbb{K}}) \subsetneq QF_{\mathbb{K}}$.*

The inclusion is strict as it was demonstrated in Example 2.4.6 that the TRS for `log` belongs to $QF_{\mathbb{Q}^+}$ and in Example 2.4.11 that it does not belong to $QI_{add}^{poly}(\mathbb{K}, \geq_{\mathbb{K}})$.

As a corollary, we obtain the intensional result that quasi-friendly programs terminating by RPO are at least as expressive as programs admitting an additive and polynomial quasi-interpretation and terminating by RPO.

Theorem 2.4.8 ([MP09]). *For $\mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+, \mathbb{R}^+\}$, for any $A \subseteq \{l\}$, $QI_{add}^{poly}(\mathbb{K}, \geq_{\mathbb{K}}) \cap RPO^A \subseteq QF_{\mathbb{K}} \cap RPO^A$.*

Notice that it is not known if the above inclusion is strict or not.

As a consequence of Theorem 2.4.8 and Theorem 2.3.5, we obtain the following negative result.

Theorem 2.4.9. *For $\mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+, \mathbb{R}^+\}$, $I_{add}^{poly}(\mathbb{K}, \geq_{\mathbb{K}}^{wf})$ and $(QF_{\mathbb{K}} \cap RPO^{\{p\}})$ are incomparable.*

DP-based characterizations capture natural algorithms that fail to be captured by RPO but it is unclear whether the converse result holds or not.

Theorem 2.4.10. *For $\mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+, \mathbb{R}^+\}$, the following statements are true:*

- $DP(TIME_{\mathbb{K}}) - (QF_{\mathbb{K}} \cap RPO^{\{p\}}) \neq \emptyset$,
- $DP(SPACE_{\mathbb{K}}) - (QF_{\mathbb{K}} \cap RPO^{\{l\}}) \neq \emptyset$,
- $SBR_{\mathbb{K}} - (QF_{\mathbb{K}} \cap RPO^{\{l\}}) \neq \emptyset$.

This is due to the fact that natural algorithms such as logarithm or greatest common divisor are captured by the above methods (see [MP08c, MP09] for more examples) whereas they are not captured by RPO due to recursive calls on sublinear computations. However, it is a difficult issue to compare the expressive power of the RPO based characterizations with the DP based ones that differ greatly in essence. Consequently, we conjecture that all these techniques are pairwise incomparable.

2.4.4 DP-interpretations for sup-interpretation synthesis

If the termination requirement is relaxed, it was shown in [MP08c] that DPs can also be used as a technique to infer sup-interpretations.

Definition 2.4.12 (DP-Interpretation). *Given a TRS $\langle \mathcal{C} \uplus \mathcal{F}, \mathcal{R} \rangle$, a (additive and polynomial) DP-Interpretation (DPI for short) is a monotonic (additive and polynomial) assignment $[-]_{\mathbb{K}}$ over \mathbb{K} extended to $\mathcal{F}^{\#}$ by $\forall \mathbf{f}, \in \mathcal{F}, [\mathbf{f}^{\#}]_{\mathbb{K}} := [\mathbf{f}]_{\mathbb{K}}$ and which satisfies:*

1. $\forall l \rightarrow r \in \mathcal{R}, [l]_{\mathbb{K}} \geq [r]_{\mathbb{K}}$,

$$2. \forall l^\sharp \rightarrow u^\sharp \in DP(\mathcal{R}), [l^\sharp]_{\mathbb{K}} \geq [u^\sharp]_{\mathbb{K}},$$

where the DP-interpretation $[-]_{\mathbb{K}}$ is extended canonically to terms as usual.

Example 2.4.12. Turning back to the TRS of Example 2.4.5, finding a DP-interpretation of the TRS consists in finding a polynomial and additive assignment $[-]_{\mathbb{K}}$ such that the following inequalities are satisfied:

$$\begin{aligned} [\text{half}(0)]_{\mathbb{K}} &\geq [0]_{\mathbb{K}}, \\ [\text{half}(1)]_{\mathbb{K}} &\geq [0]_{\mathbb{K}}, \\ [\text{half}(x+2)]_{\mathbb{K}} &\geq [\text{half}(x)+1]_{\mathbb{K}}, \\ [\log(x+2)]_{\mathbb{K}} &\geq [\log(\text{half}(x+2))+1]_{\mathbb{K}}, \\ [\log(1)]_{\mathbb{K}} &\rightarrow [0]_{\mathbb{K}}, \\ [\text{half}^\sharp(x+2)]_{\mathbb{K}} &\geq [\text{half}^\sharp(x)]_{\mathbb{K}}, \\ [\log^\sharp(x+2)]_{\mathbb{K}} &\geq [\log^\sharp(\text{half}(x+2))+1]_{\mathbb{K}}, \\ [\log^\sharp(x+2)]_{\mathbb{K}} &\geq [\text{half}^\sharp(x+2)]_{\mathbb{K}}. \end{aligned}$$

Taking the additive and polynomial assignment $[0]_{\mathbb{K}} = 0$, $[+1]_{\mathbb{K}}(X) = X + 1$, $[\text{half}]_{\mathbb{K}}(X) = X/2$, and $[\log]_{\mathbb{K}}(X) = X$, the above inequalities can be rewritten as follows:

$$\begin{aligned} 0 &\geq 0, \\ 1/2 &\geq 0, \\ X/2 + 1 &\geq X/2 + 1, \\ X + 2 &\geq X/2 + 2, \\ 1 &\rightarrow 0, \\ X/2 + 1 &\geq X/2, \\ X + 2 &\geq X/2 + 2, \\ X + 2 &\geq X/2 + 1, \end{aligned}$$

and, as they are all satisfied, $[-]_{\mathbb{K}}$ is an additive and polynomial DPI.

Let $DPI_{add}^{poly}(\mathbb{K}, \geq_{\mathbb{K}})$ be the set of function symbols whose TRS admits an additive and polynomial DP-interpretation over \mathbb{K} .

Notice that the condition on cycles in the DPG has been withdrawn and, consequently, TRS of $DPI_{add}^{poly}(\mathbb{K}, \geq_{\mathbb{K}})$ can be non-terminating. However we have the following result similar to Theorem 2.3.1.

Theorem 2.4.11 ([MP08c]). For $\mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+, \mathbb{R}^+\}$,

$$QI_{add}^{poly}(\mathbb{K}, \geq_{\mathbb{K}}) \subsetneq DPI_{add}^{poly}(\mathbb{K}, \geq_{\mathbb{K}}) \subseteq SI_{add}^{poly}(\mathbb{K}, \geq_{\mathbb{K}}).$$

As expected, the first inclusion is strict because of Example 2.4.12. Knowing whether the second inclusion is strict or not is an open issue.

We have already mentioned that the synthesis problem is undecidable for sup-interpretations in general as finding a sup-interpretation requires some prior knowledge on the termination and growth rate of the function computed by the program under study. However it was shown in [Péc13] that the DPI synthesis problem is equivalent to the quasi-interpretation synthesis problem, whose complexity is provided in Figure 1.3 for distinct function spaces, as DPIs are QIs without subterm property and with some extra inequalities based on the DPG of the TRS under study. Consequently, by Theorem 2.4.11, DPIs seem to be the good candidate at the present time to generate upper-bounds on program space consumption.

2.5 Summary

In this chapter, we have presented several results of the last decade, whose goal was to improve the expressive power of previous complexity class characterizations based on light/soft logics and interpretation methods. Moreover, we have related their expressive power, when possible.

We have studied the intensionality of criteria by comparing the set of captured programs for a fixed language and complexity class. Consequently, two criteria are often incomparable when they rely on distinct techniques.

For example, DLAL and STA are some of the most powerful techniques for light/soft logics and polynomial time but they are incomparable. In the framework of interpretations, we have $I \subsetneq QI \subsetneq DPI \subseteq SI$ and we have presented some intensional (partial) results when such techniques are mixed with termination techniques such as RPO or DP.

Following a remark by Baillot [Bai08], two criteria could also be compared by looking at their inherent complexity but this task is not straightforward. In such a framework DPI would be strictly more efficient than QI as they capture more programs and their synthesis problems are equivalent.

It is worth mentioning that one consequence of particular interest in the introduction of SI was the relaxation of the subterm property. This has improved the studies of sublinear TRSs and has led to works characterizing subpolynomial complexity classes such as `Alogtime` [BMP06] or NC^k [MP08b].

Chapter 3

Breaking the paradigm

Contents

3.1 Extensions of tiering	90
3.1.1 Imperative programs	90
3.1.2 Multi-threaded programs	94
3.1.3 Fork processes	98
3.1.4 Object oriented programs	106
3.1.5 Type inference and declassification	113
3.2 Extensions of light/soft logics	114
3.2.1 Light logic and multi-threaded programs	114
3.2.2 Soft logic and process calculi	117
3.2.3 Soft linear logic and quantum programs	119
3.2.4 Miscellaneous	121
3.3 Extensions of interpretations	122
3.3.1 Higher-order rewrite systems	122
3.3.2 Functional programs	125
3.3.3 Object oriented programs	130
3.3.4 Miscellaneous	134
3.4 Extensions of other techniques	135

In the previous chapter, we have demonstrated that ICC techniques are often difficult to compare. The main reason of this difficulty is that extensional completeness is captured but intensional completeness is sacrificed at the price of tractability. As tractability is always a reasonable requirement for an ICC technique to be effective, we come to an endless problem.

One alternative in finding new criterion with a better expressive power was to consider crossovers by trying to adapt/combine existing ICC criteria to other programming paradigms. We will discuss this interesting issue and the corresponding results obtained in the last decade in this chapter.

We start to show in Section 3.1 that tiering was successfully adapted to imperative programs including multi-threads (Subsection 3.1.2), fork process (Subsection 3.1.3), and Object-Oriented programs (Subsection 3.1.4). We discuss the type inference properties of these type systems and an extension with declassification in Subsection 3.1.5.

In Section 3.2, we focus on the extension of linear logic based approaches to other programming paradigms. We study one extension of LLL to multi-threads in Subsection 3.2.1,

one extension of SLL to a process calculus in Subsection 3.2.2, and one extension of SLL to a lambda calculus with quantum registers in Subsection 3.2.3. We discuss briefly some other extensions based on proof-nets, interaction-nets, categorical models, and realizability models in Subsection 3.2.4.

Section 3.3 presents the main extensions of the interpretation based techniques. In Subsection 3.3.1, we show how interpretations were extended to higher-order rewriting. In Subsection 3.3.2, we introduce their adaptation to a higher-order functional language and, in Subsection 3.3.3, we study their adaptation to a simple Object-Oriented programming language based on Featherweight Java [IPW01]. Several other extensions and applications are discussed in Subsection 3.3.4.

3.1 Extensions of tiering

This section is related to the extension of ramified recursion/safe recursion and, more generally, tier-based typing discipline to the imperative paradigm as initiated by the cornerstone work of Marion [Mar11]. In this section, we will survey the main results that allow the programmer to obtain polynomial time or space upper bounds on terminating and typable programs by extending the tiering technique to imperative programs [Mar11], multi-threaded programs [MP14], fork processes [HMP13], and object-oriented programs [LM13, HP15, HP18].

3.1.1 Imperative programs

In [Mar11], the leading idea is to identify the results of safe recursion [BC92] and tiering [Lei95] as a non-interference result in the context of secure flow analysis (see [SS05] for an overview of the domain). In [VIS96, SV98], Irvine, Smith, and Volpano provide a type system to certify a confidentiality policy on an imperative language and a multi-threaded language, respectively. Types are based on security levels, named High and Low. The type system prevents leak of information from level High to level Low.

In a 2-tiers based approach, the tier **0** (safe) corresponds to the High level and the tier **1** (normal) corresponds to the Low level. The type system of [Mar11] is based on an integrity policy as in [Bib77] with no read down rule rather than on a confidentiality policy as in [BLP76]. By looking at the PRN function scheme of Subsection 1.2.3 and after identifying **0** to be the type of safe data and tier **1** to be the type of normal data, one can check that data (variables) can flow from tier **1** to **0** but not the converse as illustrated below:

$$\text{PRN}(f, h_0, h_1)(\underbrace{2x + i, \bar{y}}_{\mathbf{1}}; \underbrace{\bar{z}}_{\mathbf{0}}) = h_i(\underbrace{x, \bar{y}}_{\mathbf{1}}; \underbrace{\bar{z}, \text{PRN}(f, h_0, h_1)(x, \bar{y}; \bar{z})}_{\mathbf{0}}).$$

Indeed tier **1** variables are used in a **0** position in the recursive call $\text{PRN}(f, h_0, h_1)(x, \bar{y}; \bar{z})$ and the converse never holds, *i.e.* \bar{z} is never used in a tier **1** position. It can be checked easily that this property holds for any scheme in Bellantoni and Cook's algebra (see Subsection 1.2.3).

A program P enjoys a non-interference property when for any two memory configurations (maps from variables to *values*) \mathcal{C}_1 and \mathcal{C}_2 , if the two configurations coincide on tier **1** (*i.e.* the data is the same for each variable of type **1**), P evaluates to memory configuration \mathcal{C}'_i when fed with memory configuration \mathcal{C}_i as input, for $i \in \{1, 2\}$, then \mathcal{C}'_1 and \mathcal{C}'_2 coincide on tier **1**. In other words, data of tier **0** do not have control over data of tier **1**. We demonstrate a similar non-interference result in an imperative setting which states that values stored in tier **1** variables are independent from tier **0** variables.

$$\begin{array}{c}
 \frac{}{(\mathcal{C}, \mathbf{x}) \rightarrow \mathcal{C}(\mathbf{x})} \text{ (Var)} \qquad \frac{(\mathcal{C}, \mathbf{e}_1) \rightarrow w_1 \quad \dots \quad (\mathcal{C}, \mathbf{e}_n) \rightarrow w_n}{(\mathcal{C}, \text{op}(\mathbf{e}_1, \dots, \mathbf{e}_n)) \rightarrow \llbracket \text{op} \rrbracket(w_1, \dots, w_n)} \text{ (Op)} \\
 \\
 \frac{}{(\mathcal{C}, \text{skip}) \rightarrow \mathcal{C}} \text{ (Skip)} \qquad \frac{(\mathcal{C}, \mathbf{e}) \rightarrow w}{(\mathcal{C}, \mathbf{x} := \mathbf{e}) \rightarrow \mathcal{C}[\mathbf{x} \leftarrow w]} \text{ (Asg)} \\
 \\
 \frac{(\mathcal{C}, \mathbf{c}_1) \rightarrow \mathcal{C}_1 \quad (\mathcal{C}_1, \mathbf{c}_2) \rightarrow \mathcal{C}_2}{(\mathcal{C}, \mathbf{c}_1 ; \mathbf{c}_2) \rightarrow \mathcal{C}_2} \text{ (Seq)} \qquad \frac{(\mathcal{C}, \mathbf{e}) \rightarrow w \quad (\mathcal{C}, \mathbf{c}_w) \rightarrow \mathcal{C}' \quad w \in \{1, 0\}}{(\mathcal{C}, \text{if}(\mathbf{e})\{\mathbf{c}_1\} \text{ else } \{\mathbf{c}_0\}) \rightarrow \mathcal{C}'} \text{ (Cond)} \\
 \\
 \frac{(\mathcal{C}, \mathbf{e}) \rightarrow 0}{(\mathcal{C}, \text{while}(\mathbf{e})\{\mathbf{c}\}) \rightarrow \mathcal{C}} \text{ (Wh}_0\text{)} \qquad \frac{(\mathcal{C}, \mathbf{e}) \rightarrow 1 \quad (\mathcal{C}, \mathbf{c}; \text{while}(\mathbf{e})\{\mathbf{c}\}) \rightarrow \mathcal{C}'}{(\mathcal{C}, \text{while}(\mathbf{e})\{\mathbf{c}\}) \rightarrow \mathcal{C}'} \text{ (Wh}_1\text{)}
 \end{array}$$

Figure 3.1: Big step operational semantics of imperative programs

For that purpose, we consider a simple imperative programming language computing on words of a fixed alphabet Σ , with $\{0, 1\} \subseteq \Sigma$, defined by the following grammar:

Expressions	$\mathbf{e}, \mathbf{e}_1, \dots, \mathbf{e}_n$	$::=$	$\mathbf{x} \mid \text{op}(\mathbf{e}_1, \dots, \mathbf{e}_{ar(\text{op})})$
Commands	$\mathbf{c}, \mathbf{c}_1, \mathbf{c}_2$	$::=$	$\text{skip} \mid \mathbf{x} := \mathbf{e} \mid \mathbf{c}_1 ; \mathbf{c}_2$ $\mid \text{if}(\mathbf{e})\{\mathbf{c}_1\} \text{ else } \{\mathbf{c}_2\} \mid \text{while}(\mathbf{e})\{\mathbf{c}\}$
Programs	$\mathbf{p}(\mathbf{x}_1, \dots, \mathbf{x}_n)$	$::=$	$\mathbf{c} \text{ return } \mathbf{x},$

where $\mathbf{x}, \mathbf{x}_1, \dots, \mathbf{x}_n$ are variables of the countably infinite set \mathbb{V} and op is a prefix, postfix, or infix operator of arity $ar(\text{op})$ of the countably infinite set \mathbb{O} . A memory configuration \mathcal{C} is a partial mapping from variables in \mathbb{V} to words in $\mathbb{W} = \Sigma^*$. Given a symbol a in Σ and a word w in \mathbb{W} , let $a.w$ denote the word obtained by concatenating a and w .

The semantics of the language maps a pair $(\mathcal{C}, \mathbf{c})$ consisting in a memory configuration \mathcal{C} and a command \mathbf{c} to a memory configuration \mathcal{C}' and is described in Figure 3.1, where $\llbracket \text{op} \rrbracket$ is a total function over words associated to the operator $\text{op} \in \mathbb{O}$ and $\mathcal{C}[\mathbf{x} \leftarrow w]$ is the memory configuration \mathcal{C}' equal to \mathcal{C} but on \mathbf{x} where $\mathcal{C}'(\mathbf{x}) = w$.

A program $\mathbf{p}(\mathbf{x}_1, \dots, \mathbf{x}_n) = \mathbf{c} \text{ return } \mathbf{x}$ computes the function $f : \mathbb{W}^n \rightarrow \mathbb{W}$ if for any words $w, w_1, \dots, w_n \in \mathbb{W}$:

$$(\mathcal{C}[\mathbf{x}_1 \leftarrow w_1, \dots, \mathbf{x}_n \leftarrow w_n], \mathbf{c}) \rightarrow \mathcal{C}'[\mathbf{x} \leftarrow w] \text{ if and only if } f(w_1, \dots, w_n) = w.$$

Example 3.1.1. Consider the program $\text{add}(\mathbf{x}, \mathbf{y}) = \mathbf{c} \text{ return } \mathbf{y}$ where the command \mathbf{c} is equal to

```

while( $\mathbf{x} > 0$ ){
   $\mathbf{x} := \mathbf{x} - 1$  ;
   $\mathbf{y} := \mathbf{y} + 1$ 
}
    
```

Given a unary word w , the operator > 0 tests whether it is empty or not, the operator -1 computes the predecessor, and the operator $+1$ computes the successor. These three operators can be defined formally as follows:

$$\llbracket > 0 \rrbracket(v) = \begin{cases} 1 & \text{if } \exists w \in \mathbb{W}, v = 1.w \\ 0 & \text{otherwise} \end{cases} \qquad \llbracket -1 \rrbracket(v) = \begin{cases} \epsilon & \text{if } v = \epsilon \\ u & \text{if } v = a.u, a \in \Sigma \end{cases}$$

$$\begin{array}{c}
 \frac{\Gamma(\mathbf{x}) = \alpha}{\Gamma, \Delta \vdash \mathbf{x} : \alpha} \text{ (V)} \\
 \\
 \frac{\forall i, 1 \leq i \leq n, \Gamma, \Delta \vdash \mathbf{e}_i : \alpha_i \quad \alpha_1 \rightarrow \dots \rightarrow \alpha_n \rightarrow \alpha \in \Delta(\text{op})}{\Gamma, \Delta \vdash \text{op}(\mathbf{e}_1, \dots, \mathbf{e}_n) : \alpha} \text{ (OP)} \\
 \\
 \frac{}{\Gamma, \Delta \vdash \text{skip} : \alpha} \text{ (SK)} \quad \frac{\Gamma, \Delta \vdash \mathbf{x} : \alpha \quad \Gamma, \Delta \vdash \mathbf{e} : \beta \quad \alpha \preceq \beta}{\Gamma, \Delta \vdash \mathbf{x} := \mathbf{e} : \alpha} \text{ (A)} \\
 \\
 \frac{\Gamma, \Delta \vdash \mathbf{c} : \mathbf{0}}{\Gamma, \Delta \vdash \mathbf{c} : \mathbf{1}} \text{ (SUB)} \quad \frac{\Gamma, \Delta \vdash \mathbf{e} : \alpha \quad \Gamma, \Delta \vdash \mathbf{c} : \alpha \quad \Gamma, \Delta \vdash \mathbf{c}' : \alpha}{\Gamma, \Delta \vdash \text{if}(\mathbf{e})\{\mathbf{c}\} \text{ else } \{\mathbf{c}'\} : \alpha} \text{ (C)} \\
 \\
 \frac{\Gamma, \Delta \vdash \mathbf{c} : \alpha \quad \Gamma, \Delta \vdash \mathbf{c}' : \alpha}{\Gamma, \Delta \vdash \mathbf{c} ; \mathbf{c}' : \alpha} \text{ (S)} \quad \frac{\Gamma, \Delta \vdash \mathbf{e} : \mathbf{1} \quad \Gamma, \Delta \vdash \mathbf{c} : \alpha}{\Gamma, \Delta \vdash \text{while}(\mathbf{e})\{\mathbf{c}\} : \mathbf{1}} \text{ (W)}
 \end{array}$$

Figure 3.2: Tier-based imperative type system

$$\llbracket +1 \rrbracket(v) = 1.v.$$

This program computes the unary addition. Indeed, for $n \in \mathbb{N}$, let 1^n be the unary word where the symbol 1 occurs n times. It holds that $(\mathcal{C}[\mathbf{x} \leftarrow 1^n, \mathbf{y} \leftarrow 1^m], \mathbf{c}) \rightarrow \mathcal{C}[\mathbf{x} \leftarrow \epsilon, \mathbf{y} \leftarrow 1^{m+n}]$.

We suppose given a precedence \preceq on tiers defined to be the reflexive closure of the relation \prec satisfying $\mathbf{0} \prec \mathbf{1}$. The considered tier-based type system is described in Figure 3.2. It is a subsystem of the type system of [MP14] and a simplified version of the type system of [Mar11], as it does not involve a complex lattice structure. Judgments are of the shape $\Gamma, \Delta \vdash b : \alpha$, with b an expression or command, α a tier in $\{\mathbf{0}, \mathbf{1}\}$, Γ a variable typing environment mapping each variable \mathbf{x} to a tier in $\{\mathbf{0}, \mathbf{1}\}$, and Δ an operator typing environment mapping each operator op to a set $\Delta(\text{op})$ of types of the shape $\alpha_1 \rightarrow \dots \rightarrow \alpha_{ar(\text{op})} \rightarrow \alpha$, with $\alpha_1, \dots, \alpha_{ar(\text{op})}, \alpha \in \{\mathbf{0}, \mathbf{1}\}$.

Tiers $\mathbf{0}$ and $\mathbf{1}$ have the following intuitive meaning:

- tier $\mathbf{1}$ variables will be used as guards of while loops; they should not be allowed to take more than a polynomial number of distinct values and cannot increase,
- tier $\mathbf{0}$ variables may increase and cannot be used as while loop guards.

Before stating the main non-interference and complexity results, we need to restrict the operator typing environments under consideration. Notice that considering non-restricted operator typing environments would break both results.

Definition 3.1.1. A polynomial time computable operator op is

- neutral if:
 1. either $\llbracket \text{op} \rrbracket : \mathbb{W}^{ar(\text{op})} \rightarrow \{0, 1\}$ is a predicate;
 2. or $\forall w_1, \dots, w_{ar(\text{op})} \in \mathbb{W}, \exists i \in \{1, \dots, ar(\text{op})\}, \llbracket \text{op} \rrbracket(w_1, \dots, w_{ar(\text{op})}) \preceq w_i$.

- positive if there is a constant $c_{\text{op}} \in \mathbb{N}$ such that:

$$\forall w_1, \dots, w_{\text{ar}(\text{op})} \in \mathbb{W}, \quad |\llbracket \text{op} \rrbracket(w_1, \dots, w_{\text{ar}(\text{op})})| \leq \max_i |w_i| + c_{\text{op}},$$

where \sqsubseteq is the subword relation over \mathbb{W} and the size of a word $|w|$ is equal to its number of symbols.

A neutral operator is always a positive operator but the converse is not true. In the remainder, we name positive operators those operators that are positive but not neutral.

Example 3.1.2. The operators > 0 and -1 of Example 3.1.1 defined by

$$\llbracket > 0 \rrbracket(v) = \begin{cases} 1 & \text{if } \exists w \in \mathbb{W}, v = 1.w \\ 0 & \text{otherwise} \end{cases} \quad \llbracket -1 \rrbracket(v) = \begin{cases} \epsilon & \text{if } v = \epsilon \\ u & \text{if } v = a.u, a \in \Sigma \end{cases}$$

are neutral. Indeed, > 0 computes a polynomial time computable predicate and -1 computes a subword of its input.

The $+1$ operator defined by

$$\llbracket +1 \rrbracket(v) = 1.v$$

is positive as it is not neutral and $|\llbracket +1 \rrbracket(v)| = |1.v| = |v| + 1$.

Definition 3.1.2. An operator typing environment Δ is safe if for each $\text{op} \in \text{dom}(\Delta)$, op is neutral or positive and $\forall \alpha_1 \rightarrow \dots \rightarrow \alpha_n \rightarrow \alpha \in \Delta(\text{op})$, we have:

- $\alpha \preceq \wedge_{i=1, \dots, \text{ar}(\text{op})} \alpha_i$,
- if the operator op is positive then $\alpha = \mathbf{0}$.

Definition 3.1.3 (Safe program). Given a variable typing environment Γ and an operator typing environment Δ , the program \mathbf{c} return \mathbf{x} is a safe program if there is a tier α such that $\Gamma, \Delta \vdash \mathbf{c} : \alpha$ and Δ is safe.

Example 3.1.3. Consider the program $\text{add}(\mathbf{x}, \mathbf{y})$ of Example 3.1.1. As > 0 and -1 are neutral and $+1$ is positive, the operator typing environment defined by $\Delta(\text{op}) = \{\mathbf{1} \rightarrow \mathbf{1}, \mathbf{0} \rightarrow \mathbf{0}, \mathbf{1} \rightarrow \mathbf{0}\}$, for $\text{op} \in \{> 0, -1\}$, and $\Delta(+1) = \{\mathbf{0} \rightarrow \mathbf{0}\}$ is safe.

The program can be typed as follows (we omit the environments in the judgments and one premise in the (OP) rule in order to lighten the notation).

$$\frac{\frac{\frac{\Gamma(\mathbf{x}) = \mathbf{1}}{\vdash \mathbf{x} : \mathbf{1}} (V)}{\vdash \mathbf{x} > 0 : \mathbf{1}} (OP) \quad \frac{\frac{\frac{\Gamma(\mathbf{x}) = \mathbf{1}}{\vdash \mathbf{x} : \mathbf{1}} (V) \quad \frac{\frac{\Gamma(\mathbf{x}) = \mathbf{1}}{\vdash \mathbf{x} : \mathbf{1}} (V)}{\vdash \mathbf{x} - 1 : \mathbf{1}} (OP)}{\vdash \mathbf{x} - 1 : \mathbf{1}} (A)}{\vdash \mathbf{x} := \mathbf{x} - 1 : \mathbf{1}} (A) \quad \frac{\frac{\Gamma(\mathbf{y}) = \mathbf{0}}{\vdash \mathbf{y} : \mathbf{0}} (V) \quad \frac{\frac{\Gamma(\mathbf{y}) = \mathbf{0}}{\vdash \mathbf{y} : \mathbf{0}} (V)}{\vdash \mathbf{y} + 1 : \mathbf{0}} (OP)}{\vdash \mathbf{y} + 1 : \mathbf{0}} (A)}{\vdash \mathbf{y} := \mathbf{y} + 1 : \mathbf{0}} (SUB)}{\vdash \mathbf{y} := \mathbf{y} + 1 : \mathbf{1}} (S)}{\vdash \mathbf{x} := \mathbf{x} - 1 ; \mathbf{y} := \mathbf{y} + 1 : \mathbf{1}} (W)}{\vdash \text{while}(\mathbf{x} > 0)\{\mathbf{x} := \mathbf{x} - 1 ; \mathbf{y} := \mathbf{y} + 1\} : \mathbf{1}}$$

For a given variable typing environment Γ and a given tier α , let \approx_α^Γ be an equivalence relation on memory configurations defined by $\mathcal{C} \approx_\alpha^\Gamma \mathcal{C}'$ if $\forall \mathbf{x} \in \text{dom}(\Gamma), \alpha \preceq \Gamma(\mathbf{x}) \implies \mathcal{C}(\mathbf{x}) = \mathcal{C}'(\mathbf{x})$. We are now ready to state the main non-interference and complexity results.

Theorem 3.1.1 (Non-interference). *Given a safe program c return x with respect to the typing environments Γ, Δ . For any stores \mathcal{C}_1 and \mathcal{C}_2 , if $\mathcal{C}_1 \approx_1^\Gamma \mathcal{C}_2$, $(\mathcal{C}_1, c) \rightarrow \mathcal{C}'_1$, and $(\mathcal{C}_2, c) \rightarrow \mathcal{C}'_2$, then $\mathcal{C}'_1 \approx_1^\Gamma \mathcal{C}'_2$.*

Example 3.1.4. *We have shown in Example 3.1.3 that the program of Example 3.1.1 can be typed with respect to a variable typing environment Γ such that $\Gamma(x) = \mathbf{1}$ and $\Gamma(y) = \mathbf{0}$. For $n, m, l \in \mathbb{N}$, we have $(\mathcal{C}[x \leftarrow 1^n, y \leftarrow 1^m], c) \rightarrow \mathcal{C}[x \leftarrow \epsilon, y \leftarrow 1^{m+n}]$ and $(\mathcal{C}[x \leftarrow 1^n, y \leftarrow 1^l], c) \rightarrow \mathcal{C}[x \leftarrow \epsilon, y \leftarrow 1^{n+l}]$. Moreover $\mathcal{C}[x \leftarrow 1^n, y \leftarrow 1^m] \approx_1^\Gamma \mathcal{C}[x \leftarrow 1^n, y \leftarrow 1^l]$. Consequently, by Theorem 3.1.1, $\mathcal{C}[x \leftarrow \epsilon, y \leftarrow 1^{n+m}] \approx_1^\Gamma \mathcal{C}[x \leftarrow \epsilon, y \leftarrow 1^{n+l}]$, which is obviously true.*

Theorem 3.1.2 ([Mar11, MP14]). *The set of functions over words computed by safe and terminating programs is exactly FP.*

In the above Theorem, soundness is a consequence of non-interference together with the fact that the cardinality of the set of tier $\mathbf{1}$ configurations is polynomially bounded in the input size (*i.e.* the cardinal of the space of distinct values that can be obtained as outputs of neutral operators is bounded polynomially in the size of the initial values). Consequently, if a program terminates then only a polynomial number of memory configurations distinct on tier $\mathbf{1}$ data can be encountered. Completeness is demonstrated as usual by simulating polynomials and a standard TM.

Example 3.1.5. *The program of Example 3.1.1 is safe and terminating. Consequently, it computes a function in FP.*

In what follows, let e^α , respectively $c : \alpha$, be a notation meaning that the expression e , respectively command c , is of tier α under the considered typing environments.

Example 3.1.6. *Consider the following program that computes the exponential as a counter-example.*

```

while( $x^1 > 0$ ){
   $z^? := y^0 : ?$  ;
  while( $z^? > 0$ ){
     $y^0 := y^0 + 1 : \mathbf{0}$  ;
     $z^? := z^? - 1 : ?$ 
  } ; :  $\mathbf{1}$ 
   $x^1 := x^1 - 1 : \mathbf{1}$ 
}
return y

```

It is not typable in our formalism. Indeed, suppose that it is typable. The command $y := y + 1$ enforces y to be of tier $\mathbf{0}$ since $+1$ is positive. Consequently, the command $z := y$ enforces z to be of tier $\mathbf{0}$ because of typing discipline for assignments. However, the innermost while loop enforces $z > 0$ to be of tier $\mathbf{1}$, so that z has to be of tier $\mathbf{1}$ (because $\mathbf{0} \rightarrow \mathbf{1}$ is not permitted for a safe operator typing environment) and we obtain a contradiction.

3.1.2 Multi-threaded programs

An extension to a multi-threaded language has been considered in [MP14]. A multi-threaded program $M(x_1, \dots, x_n)$, M for short, is a map from a finite set of threads identifier in $dom(M)$ to commands. In this setting, thread creation is prohibited. For a multi-threaded program M , the store \mathcal{C} plays the role of a global shared memory and is the only way for threads to communicate.

$$\begin{array}{c}
\frac{}{(\mathcal{C}, \mathbf{x}) \rightarrow_e \mathcal{C}(\mathbf{x})} \text{ (Var)} \quad \frac{(\mathcal{C}, \mathbf{e}_1) \rightarrow_e w_1 \quad \dots \quad (\mathcal{C}, \mathbf{e}_n) \rightarrow_e w_n}{\mathcal{C}, \text{op}(\mathbf{e}_1, \dots, \mathbf{e}_n) \rightarrow_e \llbracket \text{op} \rrbracket(w_1, \dots, w_n)} \text{ (Op)} \\
\\
\frac{}{(\mathcal{C}, \text{skip}; \mathbf{c}) \rightarrow_c (\mathcal{C}, \mathbf{c})} \text{ (Skp)} \quad \frac{(\mathcal{C}, \mathbf{e}) \rightarrow_e w}{(\mathcal{C}, \mathbf{x} := \mathbf{e}) \rightarrow_c (\mathcal{C}[\mathbf{x} \leftarrow w], \text{skip})} \text{ (Asg)} \\
\\
\frac{(\mathcal{C}, \mathbf{c}_1) \rightarrow_c (\mathcal{C}_1, \mathbf{c}'_1)}{(\mathcal{C}, \mathbf{c}_1; \mathbf{c}_2) \rightarrow_c (\mathcal{C}_1, \mathbf{c}'_1; \mathbf{c}_2)} \text{ (Seq)} \quad \frac{(\mathcal{C}, \mathbf{e}) \rightarrow_c w, w \in \{0, 1\}}{(\mathcal{C}, \text{if}(\mathbf{e})\{\mathbf{c}_1\} \text{ else } \{\mathbf{c}_0\}) \rightarrow_c (\mathcal{C}, \mathbf{c}_w)} \text{ (Cond)} \\
\\
\frac{(\mathcal{C}, \mathbf{e}) \rightarrow_c 0}{(\mathcal{C}, \text{while}(\mathbf{e})\{\mathbf{c}\}) \rightarrow_c (\mathcal{C}, \text{skip})} \text{ (W}_0\text{)} \quad \frac{(\mathcal{C}, \mathbf{e}) \rightarrow_c 1}{(\mathcal{C}, \text{while}(\mathbf{e})\{\mathbf{c}\}) \rightarrow_c (\mathcal{C}, \mathbf{c}; \text{while}(\mathbf{e})\{\mathbf{c}\})} \text{ (W}_1\text{)} \\
\\
\frac{M(x) = \text{skip}}{(\mathcal{C}, M) \rightarrow_{\text{nd}} (\mathcal{C}, M - x)} \text{ (NDStop)} \quad \frac{M(x) = \mathbf{c} \quad (\mathcal{C}, \mathbf{c}) \rightarrow_c (\mathcal{C}_1, \mathbf{c}_1)}{(\mathcal{C}, M) \rightarrow_{\text{nd}} (\mathcal{C}_1, M[x := \mathbf{c}_1])} \text{ (NDStep)}
\end{array}$$

Figure 3.3: Small step operational semantics of multi-threads

Non-deterministic scheduling

The small step operational semantics of multi-threads corresponds to the global relation \rightarrow_{nd} in Figure 3.3. Let $\rightarrow_{\text{nd}}^n$ be its n -fold composition. In Figure 3.3, $M - x$ is the restriction of M to $\text{dom}(M) - \{x\}$ and $M[x := \mathbf{c}]$ is a notation for the multi-threaded program M where the command assigned to x is updated to \mathbf{c} . At each step, a thread x is chosen using a fixed non-deterministic scheduling policy. Then, one step of x is performed and the control returns to the upper level. Note that the rule (*Stop*) halts the computation of a thread.

Let \emptyset be a notation for the empty multi-threaded program where all threads have terminated. A multi-threaded program $M(\mathbf{x}_1, \dots, \mathbf{x}_n)$ strongly terminates if for any store \mathcal{C} and any derivation $(\mathcal{C}, M) \rightarrow_{\text{nd}}^n (\mathcal{C}', M')$ there exist a store \mathcal{C}'' and $m \in \mathbb{N}$ such that $(\mathcal{C}', M') \rightarrow_{\text{nd}}^m (\mathcal{C}'', \emptyset)$.

The running time of a multi-threaded program $M(\mathbf{x}_1, \dots, \mathbf{x}_n)$ on inputs $w_1, \dots, w_n \in \mathbb{W}$, noted time_M , is a partial function defined by:

$$\text{time}_M(w_1, \dots, w_n) = \max\{n \mid \exists \mathcal{C}', (\mathcal{C}[\mathbf{x}_1 \leftarrow w_1, \dots, \mathbf{x}_n \leftarrow w_n], M) \rightarrow_{\text{nd}}^n (\mathcal{C}', \emptyset)\}.$$

In the special case where M strongly terminates, time_M is a total function.

The type system of Figure 3.2 can be extended to multi-threaded programs by the rule of Figure 3.4. The judgment $\Gamma, \Delta \vdash M : \diamond$ means that the multi-thread M is well-typed under the variable typing environment Γ and the operator typing environment Δ .

The notion of safe programs can be extended to safe multi-threaded programs as follows:

Definition 3.1.4. *Given Γ a variable typing environment and Δ an operator typing environment, the multi-threaded program M is safe if $\Gamma, \Delta \vdash M : \diamond$ and Δ is safe.*

$$\frac{\forall x \in \text{dom}(M), \exists \alpha \in \{\mathbf{0}, \mathbf{1}\}, \Gamma, \Delta \vdash M(x) : \alpha}{\Gamma, \Delta \vdash M : \diamond} \text{ (W)}$$

Figure 3.4: Tier-based multi-threads typing rule

Definition 3.1.5. *Let Γ be a variable typing environment and Δ be an operator typing environment.*

- *The relation $\approx_{\Gamma, \Delta}$ on commands is defined by:*
 1. *If $c_1 = c_2$ then $c_1 \approx_{\Gamma, \Delta} c_2$,*
 2. *If $\Gamma, \Delta \vdash c_1 : \mathbf{0}$ and $\Gamma, \Delta \vdash c_2 : \mathbf{0}$ then $c_1 \approx_{\Gamma, \Delta} c_2$,*
 3. *If $c_1 \approx_{\Gamma, \Delta} c_2$ and $c_3 \approx_{\Gamma, \Delta} c_4$ then $c_1; c_3 \approx_{\Gamma, \Delta} c_2; c_4$.*
- *It is extended to configurations as follows*
If $c_1 \approx_{\Gamma, \Delta} c_2$ and $\mathcal{C} \approx_1^{\Gamma} \mathcal{C}'$ then $(\mathcal{C}, c_1) \approx_{\Gamma, \Delta} (\mathcal{C}', c_2)$.
- *Finally, it is extended to multi-threads and multi-thread configurations as follows:*
 - *If $\forall x \in \text{dom}(M) = \text{dom}(M'), M(x) \approx_{\Gamma, \Delta} M'(x)$ then $M \approx_{\Gamma} M'$,*
 - *If $M \approx_{\Gamma, \Delta} M'$ and $\mathcal{C} \approx_1^{\Gamma} \mathcal{C}'$ then $(\mathcal{C}, M) \approx_{\Gamma, \Delta} (\mathcal{C}', M')$.*

We obtain a concurrent non-interference property.

Theorem 3.1.3 (Concurrent non-interference). *Let M_1 and M_2 be two safe multi-threaded programs with respect to the variable typing environment Γ and the operator typing environment Δ and let \mathcal{C}_1 and \mathcal{C}_2 be two memory configurations such that $(\mathcal{C}_1, M_1) \approx_{\Gamma, \Delta} (\mathcal{C}_2, M_2)$.*

If $(\mathcal{C}_1, M_1) \rightarrow_{\text{nd}} (\mathcal{C}'_1, M'_1)$ then there are \mathcal{C}'_2, M'_2 , and $n \in \mathbb{N}$ such that $(\mathcal{C}_2, M_2) \rightarrow_{\text{nd}}^n (\mathcal{C}'_2, M'_2)$ and $(\mathcal{C}'_1, M'_1) \approx_{\Gamma, \Delta} (\mathcal{C}'_2, M'_2)$.

It is worth noticing that the above result also holds in a sequential context when only the relation \rightarrow_c on commands is considered. Moreover, temporal variants relating the cost of evaluating **while** constructs with the rule (W₁) are also considered in [MP14].

An upper-bound on the derivation length can also be obtained in the case of strongly terminating multi-threads.

Theorem 3.1.4 ([MP14]). *Assume that $M(x_1, \dots, x_n)$ is a safe multi-threaded program that strongly terminates. There is a polynomial $Q \in \mathbb{N}[X]$ such that*

$$\forall w_1, \dots, w_n \in \mathbb{W}, \text{time}_M(w_1, \dots, w_n) \leq Q(\max_{i=1}^n (|w_i|)).$$

The soundness of Theorem 3.1.2 can be viewed as a direct application of Theorem 3.1.4 in the particular case of a multi-thread consisting in one single thread.

Example 3.1.7 ([MP14]). *Consider the following multi-thread M composed of two threads x and y computing on unary numbers.*

$$\begin{array}{ll}
x : \text{while}(x^1 > 0)^1 \{ & y : \text{while}(y^1 > 0)^1 \{ \\
\quad z^0 := z^0 + 1 : \mathbf{0} ; & \quad z^0 := 0 : \mathbf{0} ; \\
\quad x^1 := x^1 - 1 : \mathbf{1} & \quad y^1 := y^1 - 1 : \mathbf{1} \\
\} : \mathbf{1} & \} : \mathbf{1}
\end{array}$$

This program is strongly terminating. Moreover, given a store \mathcal{C} such that $\mathcal{C}(\mathbf{x}) = n$ and $\mathcal{C}(\mathbf{z}) = 0$, if $(\mathcal{C}, M) \rightarrow_{\text{nd}}^k (\mathcal{C}', \emptyset)$ then $\mathcal{C}'(\mathbf{z}) \in [0, n]$. M is safe using an operator typing environment Δ such that $\Delta(-1) = \Delta(> 0) = \{\mathbf{1} \rightarrow \mathbf{1}\}$ and $\Delta(+1) = \{\mathbf{0} \rightarrow \mathbf{0}\}$ and M strongly terminates. Consequently, by Theorem 3.1.4, it terminates in polynomial time.

Example 3.1.8 ([MP14]). Consider the following multi-thread M that shuffles two strings given as inputs.

$$\begin{array}{ll}
x : \text{while}(! (x^1 == \epsilon))^1 \{ & y : \text{while}(! (y^1 == \epsilon))^1 \{ \\
\quad z^0 := \text{cons}(\text{head}(x^1), z^0) : \mathbf{0} ; & \quad z^0 := \text{cons}(\text{head}(y^1), z^0) : \mathbf{0} ; \\
\quad x^1 := x^1 - 1 : \mathbf{1} & \quad y^1 := y^1 - 1 : \mathbf{1} \\
\} : \mathbf{1} & \} : \mathbf{1}
\end{array}$$

where cons is an operator defined by $\llbracket \text{cons} \rrbracket(\epsilon, w) = w$ and $\llbracket \text{cons} \rrbracket(a.v, w) = a.w$, that performs the concatenation of the symbol given in its first argument with its second argument. The operators $!$ and $==\epsilon$ are unary predicates and consequently can be typed by $\mathbf{1} \rightarrow \mathbf{1}$. The operator head returns the first symbol of a string given as input and can be typed by $\mathbf{1} \rightarrow \mathbf{0}$ since it is neutral. The -1 operator can be typed by $\mathbf{1} \rightarrow \mathbf{1}$ since its computation is a subterm of the input. Finally, the cons operator can be typed by $\mathbf{0} \rightarrow \mathbf{0} \rightarrow \mathbf{0}$ since $|\llbracket \text{cons} \rrbracket(u, v)| = |v| + 1$. This program is safe and strongly terminating consequently it also terminates in polynomial time.

Example 3.1.9 ([MP14]). Consider the following multi-thread M .

$$\begin{array}{ll}
x : \text{while}(x^1 > 0)^1 \{ & y : \text{while}(y^1 > 0)^1 \{ \\
\quad y^1 := x^1 ;: \mathbf{1} & \quad z^0 := z^0 + 1 ;: \mathbf{0} \\
\quad x^1 := x^1 - 1 ;: \mathbf{1} & \quad y^1 := y^1 - 1 ;: \mathbf{1} \\
\} : \mathbf{1} & \} : \mathbf{1}
\end{array}$$

Observe that, contrarily to previous examples, the guard of y depends on information flowing from x to y . Given a store \mathcal{C} such that $\mathcal{C}(\mathbf{x}) = n$, $\mathcal{C}(\mathbf{y}) = \mathcal{C}(\mathbf{z}) = 0$, if $(\mathcal{C}, M) \rightarrow_{\text{nd}}^k (\mathcal{C}', \emptyset)$ then $\mathcal{C}'(\mathbf{z}) \in [0, n \times (n + 1)/2]$. This multi-thread is safe with respect to a safe typing operator environment Δ such that $\Delta(-1) = \Delta(> 0) = \{\mathbf{1} \rightarrow \mathbf{1}\}$ and $\Delta(+1) = \{\mathbf{0} \rightarrow \mathbf{0}\}$. Moreover it strongly terminates. Consequently, it also terminates in polynomial time.

Deterministic scheduling

We extend previous results to a class of deterministic schedulers. Define $\mathcal{C} \downarrow \mathbf{1}$ as the restriction of the store \mathcal{C} to tier $\mathbf{1}$ variables. A deterministic scheduler \mathcal{S} is *quiet* if the scheduling policy depends only on the current state of the multi-threaded program M and on $\mathcal{C} \downarrow \mathbf{1}$. For example, a deterministic scheduler whose policy just depends on running threads, is quiet whereas a deterministic scheduler depending on $\mathbf{0}$ data is not quiet.

Next, we replace the two non-deterministic global transitions (NDStop) and (NDStep) of Figure 3.3 by the rules of Figure 3.5. As in previous section, the running time of a multi-threaded program $M(\mathbf{x}_1, \dots, \mathbf{x}_n)$ on inputs $w_1, \dots, w_n \in \mathbb{W}$ under the quiet scheduler \mathcal{S} , noted $\text{time}_M^{\mathcal{S}}$, is a partial function defined by:

$$\text{time}_M^{\mathcal{S}}(w_1, \dots, w_n) = n \mid \exists \mathcal{C}', (\mathcal{C}[\mathbf{x}_1 \leftarrow w_1, \dots, \mathbf{x}_n \leftarrow w_n], M) \rightarrow_{\text{d}}^n (\mathcal{C}', \emptyset).$$

In the special case where M terminates under the strategy \mathcal{S} , $\text{time}_M^{\mathcal{S}}$ is a total function.

$$\begin{array}{c}
 \frac{\mathcal{S}(M, \mathcal{C} \downarrow \mathbf{1}) = x \quad M(x) = \text{skip}}{(\mathcal{C}, M) \rightarrow_{\mathbf{d}} (\mathcal{C}, M - x)} \text{ (DStop)} \qquad \frac{\mathcal{S}(M, \mathcal{C} \downarrow \mathbf{1}) = x \quad (\mathcal{C}, M(x)) \rightarrow_c (\mathcal{C}', c')}{(\mathcal{C}, M) \rightarrow_{\mathbf{d}} (\mathcal{C}', M[x := c'])} \text{ (DStep)} \\
 \hline
 \end{array}$$

Figure 3.5: Small step operational semantics with deterministic scheduling

Theorem 3.1.5 ([MP14]). *Assume that M is a safe multi-threaded program that terminates with respect to the deterministic and quiet scheduler \mathcal{S} . There is a polynomial Q such that:*

$$\forall w_1, \dots, w_n \in \mathbb{W}, \text{time}_M^{\mathcal{S}}(w_1, \dots, w_n) \leq Q(\max_{i=1, \dots, n}(|w_i|)).$$

3.1.3 Fork processes

We now consider the extension of tiering to fork processes presented in [HMP13]. This extension requires the addition of a new tier -1 for data communication between processes and allows us to characterize the class of polynomial space computable functions **FPSPACE**.

Processes consist in programs of Section 3.1.1 extended with **fork** and **wait** constructs as follows.

Expressions	e, e_1, \dots, e_n	::=	$x \mid \text{op}(e_1, \dots, e_{ar(\text{op})})$
Commands	c, c_1, c_2	::=	$\text{fork}() \mid \text{wait}(e)$ $\mid \text{skip} \mid x := e \mid c_1 ; c_2$ $\mid \text{if}(e)\{c_1\} \text{ else } \{c_2\} \mid \text{while}(e)\{c\}$
Processes	P	::=	$c ; P \mid \text{return } x$

Each **fork** call creates a new child process with a distinct identifier (**id**) and duplicates the execution context, *i.e.* the store, including the program counter. The parent process keeps track of the **ids** of its children but not the converse. The communications between children and their parent are performed through the use of a **wait** instruction that allows a returning child to pass a value to its parent process. This programming language is simple but it is a natural fragment of a real-life programming language like **C** [KP84].

Given a store \mathcal{C} and a process P , the triplet $c = (P, \mathcal{C})_{\rho}$, where ρ is an element of $\mathcal{P}(\mathbb{N})$, is called a *configuration*. In a configuration $c = (P, \mathcal{C})_{\rho}$, the set ρ will contain the indexes of the children of the process P created during an evaluation. Let \perp be a special symbol for erased configurations.

An *environment* \mathcal{E} is a partial function from \mathbb{N} to configurations. The domain of \mathcal{E} is denoted $\text{dom}(\mathcal{E})$ and we denote $\#\mathcal{E}$ its cardinal when it is finite. We abbreviate $\mathcal{E}(n)$ by \mathcal{E}_n . The size of an environment $|\mathcal{E}|$ is defined by $|\mathcal{E}| = \sum_{i \in \text{dom}(\mathcal{E})} |\mathcal{E}_i|$. The notation $\mathcal{E}[i := c]$ is the environment \mathcal{E}' defined by $\mathcal{E}'(j) = \mathcal{E}(j)$ for all $j \neq i \in \text{dom}(\mathcal{E})$ and $\mathcal{E}'(i) = c$. As usual $\mathcal{E}[i_1 := c_1, \dots, i_k := c_k]$ is a shortcut for $\mathcal{E}[i_1 := c_1] \dots [i_k := c_k]$. The *initial* environment, noted $\mathcal{E}_{\text{init}}[P, \mathcal{C}]$, consists in the main process with no child. That is $\mathcal{E}_{\text{init}}[P, \mathcal{C}](1) = (P, \mathcal{C})_{\emptyset}$ and $\text{dom}(\mathcal{E}_{\text{init}}[P, \mathcal{C}]) = \{1\}$. An environment \mathcal{E} is *terminal* if the root process satisfies $\mathcal{E}_1 = (\text{return } x, \mathcal{C})_{\rho}$, *i.e.* the main process is returning.

The small step semantics of process configurations and environments is presented in Figure 3.6. It consists in an extension of the sequential fragment \rightarrow_c of the small step semantics of

$$\begin{aligned}
& (c ; P, \mathcal{C})_{\rho} \rightarrow (c' ; P, \mathcal{C}')_{\rho} \quad \text{if } (c, \mathcal{C}) \rightarrow_c (c', \mathcal{C}') \\
& \mathcal{E}[i := c] \rightarrow \mathcal{E}[i := c'] \quad \text{if } c \rightarrow c' \tag{Conf} \\
& \mathcal{E}[i := (x := \mathbf{fork}() ; P, \mathcal{C})_{\rho}] \rightarrow \mathcal{E}[i := (P, \mathcal{C}\{x \leftarrow \underline{n}\})_{\rho \cup \{n\}}, n := (P, \mathcal{C}\{x \leftarrow \underline{0}\})_{\emptyset}] \tag{Fork} \\
& \text{with } n = \sharp\mathcal{E} + 1 \\
& \mathcal{E}[i := (x := \mathbf{wait}(e) ; P, \mathcal{C})_{\rho}] \rightarrow \mathcal{E}[i := (P, \mathcal{C}\{x \leftarrow \mathcal{C}'(y)\})_{\rho}, n := \perp] \tag{Wait} \\
& \text{if } (e, \mathcal{C}) \rightarrow_e \underline{n}, n \in \rho \text{ and } \mathcal{E}_n = (\mathbf{return } y, \mathcal{C}')_{\rho'}
\end{aligned}$$

Figure 3.6: Small step operational semantics of environments

multi-threads presented in Figure 3.3 to configurations together with the definition of the small step semantics \rightarrow for environments.

All standard sequential commands are evaluated using rule (Conf).

The rule (Fork) creates a new configuration, a new child process, and a new store, and adds them to the environment by extending the environment domain. The parent process keeps track of the new configuration by recording its id \underline{n} inside variable x . The child id is initialized to the value $\sharp\mathcal{E} + 1$ not to conflict with other ids. Moreover, the child set of the parent configuration is updated to $\rho \cup \{n\}$.

The rule (Wait) evaluates the expression e to some binary numeral \underline{n} . If \underline{n} is equal to the id of a *terminating configuration* \mathcal{E}_n , with $n \in \rho$, (i.e. a configuration of the shape $(\mathbf{return } y, \mathcal{C}')_{\rho'}$) then the output value $\mathcal{C}'(y)$ is transmitted and stored in variable x . The returning process n is deleted by $n := \perp$.

A process P is *strongly normalizing* if there is no infinite reduction starting from the initial environment $\mathcal{E}_{init}[P, \mathcal{C}]$ through the relation \rightarrow , for any store \mathcal{C} .

Given an initial environment $\mathcal{E}_{init}[P, \mathcal{C}]$, for some strongly normalizing process P and some store \mathcal{C} , if $\mathcal{E}_{init}[P, \mathcal{C}] \rightarrow^* \mathcal{E}'$, for some environment \mathcal{E}' such that there is no environment \mathcal{E}'' , $\mathcal{E}' \rightarrow \mathcal{E}''$, then either \mathcal{E}' is a terminal configuration, i.e. $\mathcal{E}'_1 = (\mathbf{return } x, \mathcal{C}')_{\rho}$ (It means that the main process is returning), or $\mathcal{E}'_1 = (x := \mathbf{wait}(e) ; c', \mathcal{C}')_{\rho}$ (We say that the environment \mathcal{E}' is locked). A process $P = c ; \mathbf{return } x$ is *lock-free* if for any initial environment $\mathcal{E}_{init}[P, \mathcal{C}]$, there is no locked environment \mathcal{E}' such that $\mathcal{E}_{init}[P, \mathcal{C}] \rightarrow^* \mathcal{E}'$.

A process P is *confluent* if for each initial environment $\mathcal{E}_{init}[P, \mathcal{C}]$ and any two reductions $\mathcal{E}_{init}[P, \mathcal{C}] \rightarrow^* \mathcal{E}'$ and $\mathcal{E}_{init}[P, \mathcal{C}] \rightarrow^* \mathcal{E}''$ there exists an environment \mathcal{E}^3 such that $\mathcal{E}' \rightarrow^* \mathcal{E}^3$ and $\mathcal{E}'' \rightarrow^* \mathcal{E}^3$.

A strongly normalizing, lock free, and confluent process P computes a total function $f : \mathbb{W}^n \rightarrow \mathbb{W}$ defined by:

$$\forall w_1, \dots, w_n \in \mathbb{W}, f(w_1, \dots, w_n) = w$$

if $\mathcal{E}_{init}[P, \mathcal{C}[x_i \leftarrow w_i]] \rightarrow^* \mathcal{E}$, for some terminal environment \mathcal{E} with $\mathcal{E}_1 = (\mathbf{return } x, \mathcal{C}')_{\rho}$ and $\mathcal{C}'(x) = w$.

The tier based analysis can now be adapted to this programming language. In this particular setting, we need a three tier based lattice $(\{-1, \mathbf{0}, \mathbf{1}\}, \vee, \wedge)$. The induced order is such that $-1 \preceq \mathbf{0} \preceq \mathbf{1}$. Typing environments are defined in a standard way.

The new tier -1 has the following intuitive meaning: tier -1 variables will store values returned by child processes and cannot increase. They play the role of a shared memory that

$$\begin{array}{c}
\frac{\Gamma(\mathbf{x}) = \alpha}{\Gamma, \Delta \vdash \mathbf{x} : \alpha} \text{ (V)} \\
\\
\frac{\forall i, 1 \leq i \leq n, \Gamma, \Delta \vdash \mathbf{e}_i : \alpha_i \quad \alpha_1 \rightarrow \dots \rightarrow \alpha_n \rightarrow \alpha \in \Delta(\text{op})}{\Gamma, \Delta \vdash \text{op}(\mathbf{e}_1, \dots, \mathbf{e}_n) : \alpha} \text{ (OP)} \\
\\
\frac{}{\Gamma, \Delta \vdash \text{skip} : \alpha} \text{ (SK)} \quad \frac{\Gamma, \Delta \vdash \mathbf{x} : \alpha \quad \Gamma, \Delta \vdash \mathbf{e} : \beta \quad \alpha \preceq \beta}{\Gamma, \Delta \vdash \mathbf{x} := \mathbf{e} : \alpha} \text{ (A)} \\
\\
\frac{\Gamma, \Delta \vdash \mathbf{c} : \mathbf{0}}{\Gamma, \Delta \vdash \mathbf{c} : \mathbf{1}} \text{ (SUB)} \quad \frac{\Gamma, \Delta \vdash \mathbf{e} : \alpha \quad \Gamma, \Delta \vdash \mathbf{c} : \alpha \quad \Gamma, \Delta \vdash \mathbf{c}' : \alpha}{\Gamma, \Delta \vdash \text{if}(\mathbf{e})\{\mathbf{c}\} \text{ else } \{\mathbf{c}'\} : \alpha} \text{ (C)} \\
\\
\frac{\Gamma, \Delta \vdash \mathbf{c} : \alpha \quad \Gamma, \Delta \vdash \mathbf{c}' : \alpha}{\Gamma, \Delta \vdash \mathbf{c} ; \mathbf{c}' : \alpha} \text{ (S)} \quad \frac{\Gamma, \Delta \vdash \mathbf{e} : \mathbf{1} \quad \Gamma, \Delta \vdash \mathbf{c} : \alpha}{\Gamma, \Delta \vdash \text{while}(\mathbf{e})\{\mathbf{c}\} : \mathbf{1}} \text{ (WH)} \\
\\
\frac{\Gamma, \Delta \vdash \mathbf{x} : \mathbf{0}}{\Gamma, \Delta \vdash \mathbf{x} := \text{fork}() : \mathbf{0}} \text{ (F)} \quad \frac{\Gamma, \Delta \vdash \mathbf{e} : \mathbf{0} \quad \Gamma, \Delta \vdash \mathbf{x} : -\mathbf{1}}{\Gamma, \Delta \vdash \mathbf{x} := \text{wait}(\mathbf{e}) : -\mathbf{1}} \text{ (W)} \\
\\
\frac{\Gamma, \Delta \vdash \mathbf{c} : \alpha \quad \alpha \preceq \beta}{\Gamma, \Delta \vdash \mathbf{c} : \beta} \text{ (SUB)} \quad \frac{\Gamma, \Delta \vdash \mathbf{x} : \alpha}{\Gamma, \Delta \vdash \text{return } \mathbf{x} : \alpha} \text{ (RET)}
\end{array}$$

Figure 3.7: Tier-based processes type system

cannot accumulate data.

Typing rules for processes consist in the rules of Figure 3.7. One can check that the 8 first rules are similar to the rules of Figure 3.2. Consequently, the type system of Figure 3.7 is a strict extension of the type system of Figure 3.2.

As in previous type systems, the typing discipline precludes values from flowing from tier α to tier β , whenever $\alpha \preceq \beta$. The (F) typing rule enforces the tier of the variable storing the process id to be of tier $\mathbf{0}$ since the value stored will increase dynamically during the process execution. Finally, in rule (W), the tier of the variable storing the result returned by a child process has to be of tier $-\mathbf{1}$, which means that no information may flow from a variable of a child process to tier $\mathbf{0}$ and tier $\mathbf{1}$ variables of its parent process.

Again, we need to restrict the considered operators in the operator typing environment Δ . We extend the notions of neutral and positive operators of Definition 3.1.1 with the notions of max and polynomial operators.

Definition 3.1.6. *A polynomial time computable operator op is*

1. neutral if:

- (a) either $\llbracket \text{op} \rrbracket : \mathbb{W}^{\text{ar}(\text{op})} \rightarrow \{0, 1\}$ is a predicate,
 (b) or $\forall w_1, \dots, w_{\text{ar}(\text{op})} \in \mathbb{W}$, $\exists i \in \{1, \dots, \text{ar}(\text{op})\}$, $\llbracket \text{op} \rrbracket(w_1, \dots, w_{\text{ar}(\text{op})}) \leq w_i$.
2. max if for all $w_1, \dots, w_{\text{ar}(\text{op})}$, $|\llbracket \text{op} \rrbracket(w_1, \dots, w_{\text{ar}(\text{op})})| \leq \max_{i \in [1, \text{ar}(\text{op})]} |w_i|$.
3. positive if there is a constant c_{op} such that:

$$\forall w_1, \dots, w_{\text{ar}(\text{op})} \in \mathbb{W}, |\llbracket \text{op} \rrbracket(w_1, \dots, w_{\text{ar}(\text{op})})| \leq \max_{i \in [1, \text{ar}(\text{op})]} |w_i| + c_{\text{op}}.$$

4. polynomial if there is a polynomial Q such that for all $w_1, \dots, w_{\text{ar}(\text{op})}$,

$$|\llbracket \text{op} \rrbracket(w_1, \dots, w_{\text{ar}(\text{op})})| \leq Q(\max_{i \in [1, \text{ar}(\text{op})]} |w_i|).$$

Again a neutral operator is a max operator, a max operator is a positive operator, and a positive operator is a polynomial operator.

Example 3.1.10. We illustrate this classification by the operators `or`, `dis`, and `calloc` (from the `malloc` family) computing respectively the Boolean disjunction, the disjunction on binary words, and a word of size $n \times m$ on inputs of size n and m .

(Neutral)	$\llbracket \text{or} \rrbracket(u, w)$	$= 1$ $= 0$	if $u = 1$ or $w = 1$ otherwise
(Max)	$\llbracket \text{dis} \rrbracket(u_1, u_2)$	$= \llbracket \text{or} \rrbracket(a_1, a_2) \cdot \llbracket \text{dis} \rrbracket(w_1, w_2)$ $= u_{(i+1)\%2}$	if $u_i = a_i \cdot w_i$, $a_i \in \Sigma$ if $u_i = \varepsilon$, $i \in \{1, 2\}$
(Polynomial)	$\llbracket \text{calloc} \rrbracket(u, w)$	$= \underbrace{w \cdots w}_{ u \text{ times}}$	if $u, w \in \mathbb{W}$

The operator `or` is neutral because it computes a Boolean predicate in \mathbb{P} . (see Definition 3.1.1). The operator `dis` is max since it satisfies Item 2 of Definition 3.1.6. Finally, `calloc` is neither neutral, nor positive using the same reasoning and is polynomial by setting $P_{\text{calloc}}(n) = n^2$ in Item 4 of Definition 3.1.6 since $\forall w \in \mathbb{W}$, $|\llbracket \text{calloc} \rrbracket(w)| = |w| \times |w| = P_{\text{calloc}}(|w|)$.

We extend the notion of safe operator typing environment of Definition 3.1.2 to these new classes of operators.

Definition 3.1.7. An operator typing environment Δ is safe if there exist four disjoint classes of operators `Ntr`, `Max`, `Pos`, and `Pol` such that for any operator `op` and $\forall \alpha_1 \rightarrow \dots \rightarrow \alpha_n \rightarrow \alpha \in \Delta(\text{op})$, the following conditions hold:

- $\alpha \preceq \bigwedge_{i=1, n} \alpha_i$,
- if `op` \in `Ntr` then `op` is a neutral operator,
- if `op` \in `Max` then `op` is a max operator and $\alpha \neq \mathbf{1}$,
- if `op` \in `Pos` then `op` is a positive operator and $\alpha = \mathbf{0}$,
- if `op` \in `Pol` then `op` is a polynomial operator, $\alpha = \mathbf{0}$, and $\forall i$, $\alpha_i = \mathbf{1}$.

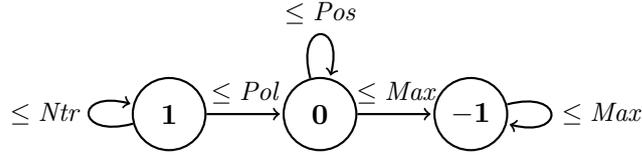


Figure 3.8: Admissible types for unary operators

We define an order relation \leq on classes of operators $Ntr \leq Max \leq Pos \leq Pol$. For $X \in \{Ntr, Max, Pos, Pol\}$, let $\leq X$ denote the set $\{Y \mid Y \leq X\}$. The constraints on operator types are summed up in Figure 3.8 for operators of arity 1.

The key point is that the type system guarantees that information flow goes from tier **1** to tier **-1**. Let us briefly explain the intuitions that lie behind such a definition. Operator types are all non-increasing wrt \preceq . Neutral operators can be iterated. A unary neutral operator can be typed by $\mathbf{1} \rightarrow \mathbf{1}$ and, consequently, can be used in the guard of a while-loop. Max operators cannot be iterated since the number of words bounded by a max function is exponential in the size. Positive operators cannot be iterated because they may increase their argument, thus leading to divergence. However they can be composed, which means that they can be typed by $\mathbf{0} \rightarrow \mathbf{0}$ and can be used in a while-loop command with no restriction. Finally, polynomial operators cannot be iterated and cannot be composed. They have to be typed by $\mathbf{1} \rightarrow \mathbf{0}$ in order to prevent composition (and, a fortiori, iteration).

Example 3.1.11. *The operators `==` and `-1` are neutral. Consequently, `-1` can have the following types $\{-1 \rightarrow -1, \mathbf{0} \rightarrow -1, \mathbf{0} \rightarrow \mathbf{0}, \mathbf{1} \rightarrow \mathbf{1}, \mathbf{1} \rightarrow \mathbf{0}, \mathbf{1} \rightarrow -1\}$ and `==` will have type in $\{\alpha \rightarrow \beta \rightarrow \gamma \mid \gamma \preceq \alpha \wedge \beta\}$. `dis` is a max operator since the disjunction of two words has size bounded by the maximal input size. Notice that `dis` $\notin Ntr$ since `dis` does not compute a subword, so it admits any type in $\{\alpha \rightarrow \beta \rightarrow \gamma \mid \gamma \preceq \alpha \wedge \beta\} - \{\mathbf{1} \rightarrow \mathbf{1} \rightarrow \mathbf{1}\}$. The operator `+1` is positive and, consequently, we may have $\Delta(+1) = \{\mathbf{0} \rightarrow \mathbf{0}\}$ if `+1` $\in Pos$ or $\Delta(+1) = \{\mathbf{1} \rightarrow \mathbf{0}\}$ if `+1` $\in Pol$. Finally, we have $\Delta(\text{calloc}) = \{\mathbf{1} \rightarrow \mathbf{1} \rightarrow \mathbf{0}\}$ since `calloc` is polynomial.*

Definition 3.1.8 (Safe process). *A process P is a safe if there are a variable typing environment Γ , a safe operator typing environment Δ , and a tier β such that $\Gamma, \Delta \vdash P : \beta$.*

The algorithm of Example 3.1.12, searching for a character in a given string and returning a Boolean number, illustrates the notion of safe process. In this example, all operators are in *Ntr* apart from `length` which is in *Pos*. Note that, for the operator `-` to be neutral, we need to consider unary numbers.

Example 3.1.12 ([HMP13]). *The following process splits its input string `str` in two parts, and performs the search of the character `*` using two forks and the operators `==` and `!` over characters or strings, `-`, and `> 0` over numbers, and Boolean disjunction `or`; the operator `getchar` such that the evaluation of `getchar(str, i)` computes the *i*-th character of the string `str`; the operator `tail(str)` which returns the string `str` minus its first symbol; and the operator `length(str)` which computes the length of the string `str`.*

```

found-1 := 0-1 : -1 ;
s1 := str1 : 1 ;
l0 := length(s1) : 0 ;
n0 := l0 : 0 ;
x0 := fork()0 : 0 ;
    
```

```

while (s != " ")1{
  if(x > 0)0{
    c0 := getchar(str1, l1)1 : 0
  } else {
    c0 := getchar(str1, n - l1)1 : 0
  }
  if(c == '*')0{
    found-1 := 1-1 : 0
  } else {skip} : 0 ;
  l0 := l - 1 : 0 ;
  s1 := tail(tail(s))1 : 1
} : 1
if(x > 0)0{
  sonf-1 := wait(x0)-1 : 0 ;
  found-1 := or(found, sonf)-1 : 0
} else {skip} : 0 ;
return found-1

```

This process is safe as it can be typed with respect to the type annotations provided above under a variable typing environment Γ such that $\Gamma(\mathbf{n}) = \Gamma(\mathbf{l}) = \Gamma(\mathbf{str}) = \mathbf{1}$, $\Gamma(\mathbf{x}) = \Gamma(\mathbf{c}) = \mathbf{0}$, and $\Gamma(\mathbf{found}) = \Gamma(\mathbf{sonf}) = -\mathbf{1}$ and under a safe operator typing environment Δ s.t. $\Delta(\mathbf{tail}) = \{\mathbf{1} \rightarrow \mathbf{1}\}$, $\Delta(-) = \Delta(\mathbf{getchar}) = \{\mathbf{1} \rightarrow \mathbf{1} \rightarrow \mathbf{1}\}$, $\Delta(==) = \Delta(!) = \{\mathbf{1} \rightarrow \mathbf{1} \rightarrow \mathbf{1}, \mathbf{0} \rightarrow \mathbf{0} \rightarrow \mathbf{0}\}$, $\Delta(\mathbf{or}) = \{-\mathbf{1} \rightarrow -\mathbf{1} \rightarrow -\mathbf{1}\}$, $\Delta(> 0) = \{\mathbf{0} \rightarrow \mathbf{0}, \mathbf{1} \rightarrow \mathbf{1}\}$ and $\Delta(\mathbf{length}) = \{\mathbf{1} \rightarrow \mathbf{0}\}$.

The `wait(x)` instruction enforces the type of the variable `found` to be $-\mathbf{1}$ using rule (W). Commands of tier $-\mathbf{1}$ can be used in a branching statement of tier $\mathbf{0}$ thanks to the subtyping rule (SUB). The `return` statement disrupts the information flow and may be considered as a declassification mechanism in the type security paradigm [VIS96].

Example 3.1.13 ([HMP13]). As another example of a fork process, let us program the counting of a character `*` in a string, which is the canonical example of distributed algorithms using MapReduce.

```

s1 := str1 : 1 ;
r0 := 1 : 0 ;
b-1 := ul(str1) : -1 ;
f-1 := 0 : -1 ;
flag0 := tt : -1 ;
while(s1 != " ")1{
  if(flag0)0{
    pidl0 := fork() : 0
    if(pidl > 0)0{
      r0 := 2 × r + 1 : 0 ;
      pidr0 := fork() : 0
    } else {
      r0 := 2 × r : 0
    }
  }
  if(or((pidl == 0), (pidr == 0))0)0{
    if(getchar(str, r) == '*')0{
      f-1 := addmod(f-1, 1, b-1) : 0
    } else {skip}
  } else {
    flag0 := ff : 0 ;
    xl-1 := wait(pidl) : 0 ;
    xr-1 := wait(pidr) : 0 ;
  }
}

```

```

        f-1 := addmod(f-1, x1-1, b-1) : 0 ;
        f-1 := addmod(f-1, xr-1, b-1) : 0
    }
}
s1 := half(s)1 : 1
};
return f

```

`or`, `!=`, `==`, and `getchar(str, i)` are the neutral operators described in Example 3.1.12. `half` is returning the first half of a string. It is also neutral. `ul(str)` stands for unary length, it is the binary number containing as many 1 as there are characters in `str`. This operator is positive. `2×` and `+1` are the binary numerical operators. They are positive. `addmod(x,y,b)` computes the addition of `x` and `y` modulo `b`. It is in *Max*. It is straightforward to verify that this process is safe with respect to the type annotations provided above.

Now we are ready to state the main result, a characterization of polynomial space computable functions.

Theorem 3.1.6 ([HMP13]). *The set of functions computed by strongly normalizing, lock-free, confluent, and safe processes is exactly FPSPACE.*

Soundness is achieved as follows: the type discipline still precludes flows from lower levels to higher level. Data of tier **1** does not increase and control while loops. Data of tier **0** can increase at most polynomially in each process. Because of the (Fork) typing rule, ids are of tier **0** and, consequently, there cannot be more than a polynomial number of created processes in depth (where depth is the childhood relation). However the total number of processes can be exponential. The polynomial upper bound on computations is recovered by the type restriction on communication data: they are of tier **-1** and only max operators can be applied. Hence no data accumulation is allowed between processes.

Completeness is shown by simulating a PSPACE-complete problem: QBF by a strongly normalizing and safe process. Consequently, we can compute the *i*-th output bit of each FPSPACE function as illustrated by Example 3.1.14.

Example 3.1.14 ([HMP13]). *The following strongly normalizing and safe process computes QBF on a formula `phi` given as input. It generates 2^n forks, *n* being the number of variables in `phi`, each of these processes is responsible for computing `phi` on a fixed Boolean assignment. This process will use the following operators for which we provide a safe operator typing environment Δ :*

- `fst`, `snd`, `tail` and `rmd` respectively giving the first element, the second element, the word without its first character, and the word without its two first characters:
 $\Delta(\text{fst}) = \Delta(\text{snd}) = \Delta(\text{tail}) = \Delta(\text{rmd}) = \{\mathbf{0} \rightarrow \mathbf{0}, \mathbf{1} \rightarrow \mathbf{1}\}$.
- `cons` a positive operator adding a head character to a word: $\Delta(\text{cons}) = \{\mathbf{1} \rightarrow \mathbf{0} \rightarrow \mathbf{0}\}$.
- the neutral Boolean operators `or` and `and` and the neutral operator `evaluate` which evaluates a Boolean formula with respect to an evaluation encoded as an array of Boolean numbers.
- `calloc` which builds an array; `read` and `write` for accessing this array.

`tab := calloc(Y, Z)` allocates a table of size $|Y| \times |Z|$ (for storing $|Y|$ elements of size at most $|Z|$) in which we can read with `v := read(tab, i, Z)` which queries the word `tab[i|Z| : (i+1)|Z|]` and we can write using `status := write(tab, v, i, Z)` which

writes word v in tab between positions $i|Z|$ and $(i+1)|Z|$. $\Delta(\text{calloc}) = \{1 \rightarrow 1 \rightarrow 0\}$.
 $\Delta(\text{read}) = \{0 \rightarrow 0 \rightarrow 1 \rightarrow 0\}$, $\Delta(\text{write}) = \{0 \rightarrow 0 \rightarrow 0 \rightarrow 1 \rightarrow 0\}$.

The operators > 0 , $==$, fst , snd , tail , rmd , or , and , read , write , and evaluate are neutral. Consequently, we can set $-1 \rightarrow -1 \rightarrow -1 \in \Delta(\text{or}) \cap \Delta(\text{and})$. The operators cons , -1 , and $+1$ are positive and the operator calloc is polynomial.

```

psi1 := phi1 : 1 ;
q1 := fst(phi1) : 1 ;
pidtab0 := calloc(phi1, phi1)0 : 0 ;
vartab0 := calloc(phi1, tt1)0 : 0 ;
i0 := 00 : 0 ;
v1 := snd(phi1)1 : 1 ;
while(or(q1 == '∃'), (q1 == '∀'))1{
  phi1 := rmd(phi1) : 1 ;
  pid0 := fork() : 0 ;
  if(pid0 > 0){
    status0 := write(pidtab, pid, i, phi1) : 0 ;
    status0 := write(vartab, tt0, v, tt) : 0 ;
    i0 := i + 10 : 0
  } else {
    opstack0 := ε0 : 0 ;
    status0 := write(vartab, ff, v, tt) : 0 ;
    i0 := 00 : 0
  }
  if(q1 == '∃')1{
    opstack0 := cons('∨', opstack0) : 1
  } else {
    opstack0 := cons('∧', opstack0) : 1 ;
  }
  q1 := fst(phi1) : 1 ;
  v1 := snd(phi1) : 1
}
res-1 := evaluate(phi1, vartab)0 : 0 ;
q1 := fst(psi1) : 1 ;
while(or((q1 == '∃'), (q1 == '∀'))1{
  phi1 := rmd(phi1) : 1 ;
  q1 := fst(psi1) : 1 ;
  i0 := i - 10 : 0 ;
  pid0 := read(pidtab0, i, phi1) : 0 ;
  op0 := fst(opstack0) : 0 ;
  opstack0 := tail(opstack0) : 0 ;
  resson-1 := wait(pid0)-1 : -1
  if(op0 == '∨')1{
    res-1 := or(res, resson)-1 : 0 ;
  } else {
    res-1 := and(res, resson)-1 : 0 ;
  }
}
} ;
return res-1

```

This process satisfies the lock-freedom and confluence properties. Consequently, we can compute the i -th output bit of each polynomial space computable function since we can reduce any polynomial space decision problem to QBF using a process without forks. Now it remains to

show that all these results can be combined in order to compute the whole function. The decision problem $\{(x, y) \mid f(x) = y \text{ and } f \in \text{FPSPACE}\}$ is in PSPACE. So, in order to compute $f(x)$, we first generate all possible words y of the right size and then we test whether or not (x, y) is in the graph of f using a fork process for each case.

3.1.4 Object oriented programs

In [LM13], ramification and tiering were extended to a programming language over a dynamical graph structure, a structure where edges and vertices can be created, deleted, and updated. This paper provides a characterization of polynomial time on such programs. It has been extended in [HP15, HP18] to the study of Object-Oriented Java-like programs including recursive methods. [HP18] provides a more general treatment of recursive methods than [HP15] as they can compute increasing data. This improvement is handled by restricting the contexts under which such methods can be called.

The syntax of considered programs is defined by the following grammar:

$$\begin{aligned}
 \text{Expressions } \ni e & ::= x \mid \text{cst}_\tau \mid \text{null} \mid \text{this} \mid \text{op}(\bar{e}) \mid \text{new } C(\bar{e}) \mid e.m(\bar{e}) \\
 \text{Instructions } \ni I & ::= ; \mid [\tau] x := e; \mid I_1 I_2 \mid \text{while}(e)\{I\} \mid \text{if}(e)\{I_1\}\text{else}\{I_2\} \mid e.m(\bar{e}); \\
 \text{Methods } \ni m_C & ::= \tau m(\overline{\tau x})\{I[\text{return } x;]\} \\
 \text{Constructors } \ni k_C & ::= C(\overline{\tau y})\{I\} \\
 \text{Classes } \ni C & ::= C [\text{extends } D] \{\overline{\tau x}; \overline{k_C} \overline{m_C}\},
 \end{aligned}$$

where $x \in \mathbb{V}$, $\text{op} \in \mathbb{O}$, $m \in \mathbb{M}$, and $C \in \mathbb{C}$. The four disjoint sets \mathbb{V} , \mathbb{O} , \mathbb{M} , and \mathbb{C} represent the set of variables, the set of operators, the set of method names, and the set of class names, respectively. In the above grammar, $[e]$ denotes some optional syntactic element e and \bar{e} denotes a sequence of syntactic elements e_1, \dots, e_n . \mathbb{C} is the set of class names C and \mathbb{T} is the set of primitive data types τ , including $\{\text{void}, \text{boolean}, \text{int}, \text{char}\}$. The metavariable cst_τ represents a primitive type constant of type $\tau \in \mathbb{T}$. Each primitive operator $\text{op} \in \mathbb{O}$ has a fixed arity n and comes equipped with a signature of the shape $\text{op} :: \tau_1 \times \dots \times \tau_n \rightarrow \tau_{n+1}$ fixed by the language implementation. Given a method $\tau m(\tau_1 x_1, \dots, \tau_n x_n)\{I [\text{return } x;]\}$ of C , its signature is $\tau m^C(\tau_1, \dots, \tau_n)$, the notation m^C denoting that m is declared in C . The signature of a constructor k_C is $C(\overline{\tau})$.

Variables are split into local variables, fields and parameters. In what follows, let $C.\mathcal{F}$ be the set of field names of the class C .

The fields of a class instance can only be accessed through the use of getters, *i.e.* cannot be accessed directly using the “.” operator. Consequently, all fields are implicitly **private**. On the opposite, methods and classes are all implicitly **public**.

Inheritance (and override) is allowed through the use of the **extends** construct. If C **extends** D , the constructors $C(\overline{\tau y})\{I\}$ are constructors initializing both the fields of C and the fields of D . Inheritance defines a partial order on classes denoted by $C \trianglelefteq D$.

A *program* is a collection of classes including exactly one class $\text{Exe}\{\text{void main}()\{\text{Init Comp}\}\}$, with $\text{Init}, \text{Comp} \in \text{Instructions}$. The method **main** of the class **Exe** is intended to be the entry point of the program. The instruction **Init** is called the *initialization instruction*. Its purpose is to compute the program input, which is strongly needed in order to define the complexity of a given program since, without input, all terminating programs would be considered to be constant time programs. The instruction **Comp** is called the *computational instruction*. The type system presented below will analyze the complexity of this latter instruction.

We restrict the considered programs to well-formed programs satisfying the following conditions. There is only one class per class name. A local variable \mathbf{x} is declared and initialized exactly once by a $\tau \mathbf{x} := \mathbf{e}$; instruction. A method output type is `void` when it has no `return` statement. Each signature is unique.

The operational semantics of this programming language is fully described in [HP18]. It relates pairs of memory configuration and an instruction. In this particular setting, a memory configuration is both a heap, describing the state of global variables, and a stack, keeping information on method calls and parameter values.

A *tiered type* is a pair $\tau(\alpha)$ consisting of a type $\tau \in \mathbb{C} \cup \mathbb{T}$ together with a tier $\alpha \in \{\mathbf{0}, \mathbf{1}\}$. Given a sequence of types $\bar{\tau} = \tau_1, \dots, \tau_n$, a sequence of tiers $\bar{\alpha} = \alpha_1, \dots, \alpha_n$, and a tier α , let $\bar{\tau}(\bar{\alpha})$ denote $\tau_1(\alpha_1), \dots, \tau_n(\alpha_n)$, $\bar{\tau}(\alpha)$ denote $\tau_1(\alpha), \dots, \tau_n(\alpha)$, and $\langle \bar{\tau} \rangle$ ($\langle \bar{\tau}(\bar{\alpha}) \rangle$, respectively) denote the Cartesian product of types (tiered types, respectively).

Before introducing the considered type system, we introduce four other kinds of environments:

- *operator typing environments* Δ defined as usual,
- *method typing environments* δ associating a tiered type to each program variable $v \in \mathbb{V}$,
- *typing environments* Ω associating a method typing environment δ to each method signature $\tau \mathbf{m}^c(\bar{\tau})$, *i.e.* $\Omega(\tau \mathbf{m}^c(\bar{\tau})) = \delta$,
- *contextual typing environments* $\Gamma = (s, \Omega)$, a pair consisting of a method signature and a typing environment; the method (or constructor) signature s indicates under which context (typing environment) the fields are typed.

For each $\mathbf{x} \in \mathbb{V}$, define $\Gamma(\mathbf{x}) = \Omega(s)(\mathbf{x})$. Also define $\Gamma\{\mathbf{x} \leftarrow \tau(\alpha)\}$ to be the contextual typing environment Γ' such that $\forall \mathbf{y} \neq \mathbf{x}, \Gamma'(\mathbf{y}) = \Gamma(\mathbf{y})$ and $\Gamma'(\mathbf{x}) = \tau(\alpha)$. Let $\Gamma\{s'\}$ be a notation for the contextual typing environment that is equal to (s', Ω) , *i.e.* the signature s' has been substituted to s in Γ .

The type system is presented in Figure 3.9 where, given a sequence $\bar{\mathbf{e}} = \mathbf{e}_1, \dots, \mathbf{e}_n$ of expressions, a sequence of types $\bar{\tau} = \tau_1, \dots, \tau_n$, and two sequences of tiers, $\bar{\alpha} = \alpha_1, \dots, \alpha_n$ and $\bar{\beta} = \beta_1, \dots, \beta_n$, the notation $\Gamma, \Delta \vdash_{\bar{\beta}} \bar{\mathbf{e}} : \bar{\tau}(\bar{\alpha})$ means that $\Gamma, \Delta \vdash_{\beta_i} \mathbf{e}_i : \tau_i(\alpha_i)$ holds, for all $i \in [1, n]$.

(*) The rule (Ass) enforces the following side conditions:

- if \mathbf{x} is a field then $\alpha = \mathbf{0}$,
- if $\tau \in \mathbb{T}$ then $\alpha' \leq \alpha$,
- if $\tau \in \mathbb{C}$ then $\alpha' = \alpha$.

(**) The rule (Body) requires that $\tau \mathbf{m}(\bar{\tau} \bar{\mathbf{x}})\{\mathbf{I} [\text{return } \mathbf{x};]\} \in \mathbb{C}$.

There are four kinds of typing judgments:

- $\Gamma, \Delta \vdash_{\beta} \mathbf{e} : \tau(\alpha)$ for expressions, meaning that the expression \mathbf{e} is of tiered type $\tau(\alpha)$ under the contextual typing environment Γ and operator typing environment Δ and can only be assigned to in an instruction of tier at least β ,
- $\Gamma, \Delta \vdash \mathbf{I} : \text{void}(\alpha)$ for instructions, meaning that the instruction \mathbf{I} is of tiered type `void`(α) under the contextual typing environment Γ and operator typing environment Δ ,

$$\begin{array}{c}
\frac{}{\Gamma, \Delta \vdash_{\beta} \mathbf{cst}_{\tau} : \tau(\alpha)} \text{ (Cst)} \quad \frac{}{\Gamma, \Delta \vdash_{\beta} \mathbf{null} : \mathbf{C}(\alpha)} \text{ (Null)} \\
\\
\frac{\Gamma(\mathbf{x}) = \tau(\alpha)}{\Gamma, \Delta \vdash_{\beta} \mathbf{x} : \tau(\alpha)} \text{ (Var)} \quad \frac{\Gamma, \Delta \vdash_{\bar{\beta}} \bar{\mathbf{e}} : \bar{\tau}(\alpha) \quad \langle \bar{\tau}(\alpha) \rangle \rightarrow \tau(\alpha) \in \Delta(\mathbf{op})}{\Gamma, \Delta \vdash_{\sqrt{\beta}} \mathbf{op}(\bar{\mathbf{e}}) : \tau(\alpha)} \text{ (Op)} \\
\\
\frac{\forall \mathbf{x} \in \mathbf{C.F}, \exists \tau, \Gamma(\mathbf{x}) = \tau(\alpha)}{\Gamma, \Delta \vdash_{\beta} \mathbf{this} : \mathbf{C}(\alpha)} \text{ (Self)} \quad \frac{\Gamma, \Delta \vdash_{\beta} \mathbf{e} : \mathbf{D}(\alpha) \quad \mathbf{D} \trianglelefteq \mathbf{C}}{\Gamma, \Delta \vdash_{\beta} \mathbf{e} : \mathbf{C}(\alpha)} \text{ (Pol)} \\
\\
\frac{\Gamma, \Delta \vdash_{\bar{\beta}} \bar{\mathbf{e}} : \bar{\tau}(\mathbf{0}) \quad \Gamma\{\mathbf{C}(\bar{\tau})\}, \Delta \vdash \mathbf{C}(\bar{\tau}) : \langle \bar{\tau}(\mathbf{0}) \rangle \rightarrow \mathbf{C}(\mathbf{0})}{\Gamma, \Delta \vdash_{\sqrt{\beta}} \mathbf{new} \mathbf{C}(\bar{\mathbf{e}}) : \mathbf{C}(\mathbf{0})} \text{ (New)} \\
\\
\frac{\Gamma, \Delta \vdash_{\beta} \mathbf{e} : \mathbf{C}(\alpha') \quad \Gamma, \Delta \vdash_{\bar{\beta}} \bar{\mathbf{e}} : \bar{\tau}(\bar{\alpha}) \quad \Gamma\{\tau \mathbf{m}^{\mathbf{C}}(\bar{\tau})\}, \Delta \vdash_{\beta} \tau \mathbf{m}^{\mathbf{C}}(\bar{\tau}) : \mathbf{C}(\beta) \times \langle \bar{\tau}(\bar{\alpha}) \rangle \rightarrow \tau(\alpha)}{\Gamma, \Delta \vdash_{\beta} \mathbf{e} : \mathbf{C}(\alpha)} \text{ (Call)} \\
\\
\frac{\forall i, \Gamma, \Delta \vdash \mathbf{I}_i : \mathbf{void}(\alpha_i)}{\Gamma, \Delta \vdash \mathbf{I}_1 \mathbf{I}_2 : \mathbf{void}(\alpha_1 \vee \alpha_2)} \text{ (Seq)} \quad \frac{\Gamma, \Delta \vdash \mathbf{I} : \mathbf{void}(\mathbf{0})}{\Gamma, \Delta \vdash \mathbf{I} : \mathbf{void}(\mathbf{1})} \text{ (Sub)} \\
\\
\frac{\Gamma, \Delta \vdash_{\alpha} \mathbf{e} : \mathbf{boolean}(\alpha) \quad \forall i, \Gamma, \Delta \vdash \mathbf{I}_i : \mathbf{void}(\alpha)}{\Gamma, \Delta \vdash \mathbf{if}(\mathbf{e})\{\mathbf{I}_1\}\mathbf{else}\{\mathbf{I}_2\} : \mathbf{void}(\alpha)} \text{ (If)} \\
\\
\frac{\Gamma, \Delta \vdash_{\mathbf{1}} \mathbf{e} : \mathbf{boolean}(\mathbf{1}) \quad \Gamma, \Delta \vdash \mathbf{I} : \mathbf{void}(\mathbf{1})}{\Gamma, \Delta \vdash \mathbf{while}(\mathbf{e})\{\mathbf{I}\} : \mathbf{void}(\mathbf{1})} \text{ (Wh)} \\
\\
\frac{[\Gamma, \Delta \vdash_{\mathbf{0}} \mathbf{x} : \tau(\alpha')] \quad \Gamma, \Delta \vdash_{\beta} \mathbf{e} : \tau(\alpha)}{\Gamma, \Delta \vdash [[\tau] \mathbf{x} :=] \mathbf{e} : \mathbf{void}(\alpha \vee \beta)} \text{ (Ass*)} \\
\\
\frac{\Gamma\{\bar{\mathbf{x}} \leftarrow \bar{\tau}(\mathbf{0})\}, \Delta \vdash \mathbf{I} : \mathbf{void}(\mathbf{0}) \quad \mathbf{C}(\bar{\tau} \bar{\mathbf{x}})\{\mathbf{I}\} \in \mathbf{C}}{\Gamma, \Delta \vdash \mathbf{C}(\bar{\tau}) : \langle \bar{\tau}(\mathbf{0}) \rangle \rightarrow \mathbf{C}(\mathbf{0})} \text{ (Cons)} \\
\\
\frac{\mathbf{C} \trianglelefteq \mathbf{D} \quad \Gamma, \Delta \vdash_{\gamma} \tau \mathbf{m}^{\mathbf{D}}(\bar{\tau}) : \mathbf{D}(\alpha') \times \langle \bar{\tau}(\bar{\alpha}) \rangle \rightarrow \tau(\alpha) \quad \tau \mathbf{m}(\bar{\tau} \bar{\mathbf{x}})\{\mathbf{I} \text{ [return } \mathbf{x};]\} \in \mathbf{D}}{\Gamma, \Delta \vdash_{\gamma} \tau \mathbf{m}^{\mathbf{C}}(\bar{\tau}) : \mathbf{C}(\alpha') \times \langle \bar{\tau}(\bar{\alpha}) \rangle \rightarrow \tau(\alpha)} \text{ (OR)} \\
\\
\frac{\Gamma\{\mathbf{this} \leftarrow \mathbf{C}(\beta), \bar{\mathbf{x}} \leftarrow \bar{\tau}(\bar{\alpha}), [\mathbf{x} \leftarrow \tau(\alpha)]\}, \Delta \vdash \mathbf{I} : \mathbf{void}(\gamma) \quad \Gamma, \Delta \vdash_{\gamma} \mathbf{this} : \mathbf{C}(\beta)}{\Gamma, \Delta \vdash_{\gamma} \tau \mathbf{m}^{\mathbf{C}}(\bar{\tau}) : \mathbf{C}(\beta) \times \langle \bar{\tau}(\bar{\alpha}) \rangle \rightarrow \tau(\alpha)} \text{ (Body**) }
\end{array}$$

Figure 3.9: Tier-based OO type system

- $\Gamma, \Delta \vdash_{\beta} s : \mathbf{C}(\beta) \times \langle \bar{\tau}(\bar{\alpha}) \rangle \rightarrow \tau(\alpha)$ for method signatures, meaning that the method m of signature s belongs to the class \mathbf{C} ($\mathbf{C}(\beta)$ is the tiered type of the current object `this`), has parameters of type $\langle \bar{\tau}(\bar{\alpha}) \rangle$, has a return variable of type $\tau(\alpha)$, with $\tau = \text{void}$ in the particular case where there is no return statement, and can only be called in instructions of tier at least β ,
- $\Gamma, \Delta \vdash \mathbf{C}(\bar{\tau}) : \langle \bar{\tau}(\mathbf{0}) \rangle \rightarrow \mathbf{C}(\mathbf{0})$ for constructor signatures, meaning that the constructor \mathbf{C} has parameters of type $\langle \bar{\tau}(\mathbf{0}) \rangle$ and a return variable of type $\mathbf{C}(\mathbf{0})$, matching the class type \mathbf{C} .

A program of executable `Exe{void main(){Init Comp}}` is well-typed if there are a typing environment Ω and an operator typing environment Δ such that $(\text{void main}^{\text{Exe}}(), \Omega), \Delta \vdash \text{Comp} : \text{void}(\mathbf{1})$ can be derived.

Example 3.1.15 ([HP18]). *Consider the class of linked lists of Boolean numbers BList.*

```

BList{
  boolean value ;
  BList queue ;

  BList(boolean v,BList q){
    value := v ;
    queue := q ;
  }

  BList getQueue(){return queue ; }

  void setQueue(BList q){
    queue := q ;
  }

  boolean getValue(){return value ; }

  BList clone(){
    BList n ;
    if(value != null){
      n := new BList(value,queue.clone()) ;
    } else {
      n := new BList(null,null) ;
    }
    return n ;
  }

  void decrement(){
    if(value == true){
      value := false ;
    } else {
      if(queue != null){
        value := true ;
        queue.decrement() ;
      } else {
        value := false ;
      }
    }
  }
}

```

```

int length(){
    int res := 1 ;
    if(queue != null){
        res := queue.length() ;
        res := res + 1 ;
    } else { ; }
    return res ;
}
}

```

We are trying to type each method and constructor of this class using the type system of Figure 3.9.

```

BList(boolean v, BList q){
    value := v ;
    queue := q ;
}

```

The constructor `BList` can be typed by $\text{boolean}(\mathbf{0}) \times \text{BList}(\mathbf{0}) \rightarrow \text{BList}(\mathbf{0})$, using rules (Cons), (Seq), and twice rule (Ass). In the two applications of rule (Ass), the tiers of the fields `value` and `queue` are enforced to be $\mathbf{0}$ because of the side condition (*). As a consequence, it is not possible to create objects of tiered type $\text{BList}(\mathbf{1})$ in a computational instruction.

```

BList getQueue(){return queue ; }

```

`getQueue` can be typed by $\text{BList}(\mathbf{1}) \rightarrow \text{BList}(\mathbf{1})$ or $\text{BList}(\mathbf{0}) \rightarrow \text{BList}(\mathbf{0})$, by rules (Body) and (Self). The types $\text{BList}(\alpha) \rightarrow \text{BList}(\beta)$, $\alpha \neq \beta$, are prohibited because of rules (Body) and (Self) since the tier of the current object has to match the tier of its fields. In the same manner, the method `getValue` can be given the types $\text{BList}(\alpha) \rightarrow \text{boolean}(\alpha)$, $\alpha \in \{\mathbf{0}, \mathbf{1}\}$.

The method `setQueue`

```

void setQueue(BList q){
    queue := q ;
}

```

can only be given the types $\text{BList}(\mathbf{0}) \times \text{BList}(\mathbf{0}) \rightarrow \text{void}(\beta)$. Indeed, by rule (Ass), `queue` and `q` are enforced to be of tier $\mathbf{0}$. Consequently, this is of tier $\mathbf{0}$ by rules (Body) and (Self). The tier of the body can be $\mathbf{0}$ or $\mathbf{1}$, using subtyping rule (Sub).

The method `decrement` can be typed by the following annotations:

```

void decrement(){
    if(value0 == true){
        value := false ; :  $\mathbf{0}$ 
    } else {
        if(queue != null){
            value0 := true ;
            queue0.decrement() ; :  $\mathbf{0}$ 
        } else {
            value0 := false ; :  $\mathbf{0}$ 
        }
    }
}
}

```

Because of the rules (Ass) and (Body), it can be given the type $\text{BList}(\mathbf{0}) \rightarrow \text{void}(\mathbf{0})$. As we shall see later, this method will be rejected by the safety condition as it might lead to exponential length derivation whenever it is called in a while loop.

The method `length`

```

int length(){
  int res := 1 ; : 0
  if(queue1 != null){
    res := queue.length() ; : 1
    res := res + 1 ; : 0
  }
  else { ; }
  return res ;
}

```

can be typed by $\Gamma, \Omega \vdash_1 \text{int length}^{\text{BList}}() : \text{BList}(\mathbf{1}) \rightarrow \text{int}(\mathbf{0})$ with respect to the annotations provided above.

Let us now give some intuitions on the type system of Figure 3.9. First, as in the imperative case, data may only flow from higher tier to lower tier. However, in the rule (Ass) the side condition enforces equality of tiers whenever the assignment is on objects. Consequently, a flow from $\mathbf{1}$ to $\mathbf{0}$ can only occur on primitive data. While this could look restrictive at first glance, this restriction is natural. As object data are passed by reference (and not by value), a flow from $\mathbf{1}$ to $\mathbf{0}$ could allow tier $\mathbf{0}$ variable to access and change tier $\mathbf{1}$ data, hence breaking the non-interference property, as illustrated by the following example.

Example 3.1.16 ([HP18]). Consider the following method

```

BList extend(BList l){
  BList x := l ;
  while(l != null){
    x := new BList(true, x) ;
    l := l.getQueue() ;
  }
  return x ;
}

```

In the rule (Ass) of Figure 3.9, for the assignment $x := l$ to be typed, the tiers of x and l are required to be equal as the corresponding type is object (`BList`). Consequently, the above code cannot be typed as l is enforced to be of tier $\mathbf{1}$ in the while loop guard and x is enforced to be of tier $\mathbf{0}$ by rules (Ass) and (New) applied to subcommand $l := l.getQueue()$. This typing discipline is restrictive but prevents a tier $\mathbf{0}$ variable x from pointing to tier $\mathbf{1}$ data. Indeed this would allow to change tier $\mathbf{1}$ data structure by iterating on this tier $\mathbf{0}$ variable and, consequently, the non-interference would be broken. Notice however that the following variant with cloning can be typed

```

BList extend(BList l){
  BList x := l.clone() ;
  while(l != null){
    x := new BList(true, x) ;
    l := l.getQueue() ;
  }
  return x ;
}

```

provided that the method `clone` can be given the type $\text{BList}(\mathbf{1}) \rightarrow \text{BList}(\mathbf{0})$. This is possible as we will shortly see in Example 3.1.18. Here the clone method deep copies the tier $\mathbf{1}$ data. Hence accessing this value inside a tier $\mathbf{0}$ variable does not break the non-interference.

The type system of Figure 3.9 is by nature not restrictive enough for handling recursive calls in polynomial time. Indeed two recursive calls can be combined or accumulated in the body of a recursive method, allowing to compute, for example, exponential time functions such as the Fibonacci sequence. Moreover, the nesting of a while loop in the body of a recursive method can lead to exponential behavior by allowing dynamically nested while loops. Worst of all, non-interference does not hold for such methods. Indeed, a recursive method call assigned to a tier $\mathbf{0}$ variable could be controlled by a tier $\mathbf{0}$ expression.

Now we put some aside restrictions on recursive methods to ensure that their computations remain polynomially bounded by preventing the above issues.

Definition 3.1.9 (Safe OO program). *A well-typed program with respect to a typing environment Ω and operator typing environment Δ is safe if for each recursive method $\tau \mathbf{m}(\overline{\tau \mathbf{x}})\{\mathbf{I} [\mathbf{return} \mathbf{x};]\}$:*

1. *there is exactly one call to the recursive method (or any mutually recursive method) in the instruction \mathbf{I} ,*
2. *there is no while loop inside \mathbf{I} , and*
3. *only judgments of the shape $(s, \Omega), \Delta \vdash_{\mathbf{1}} \tau \mathbf{m}^{\mathbf{C}}(\overline{\tau \mathbf{x}}) : \mathbf{C}(\mathbf{1}) \times \langle \overline{\tau(\mathbf{1})} \rangle \rightarrow \tau(\alpha)$ can be derived using rules (Body) and (OR).*

Example 3.1.17. *Consider a well-typed program whose computational instruction calls the methods `getQueue`, `getValue`, `setQueue`, or `length` of Example 3.1.15. This program is safe. Indeed, the only recursive method is `length`. As illustrated in Example 3.1.15, it can be typed by $\mathbf{BList}(\mathbf{1}) \rightarrow \mathbf{int}(\mathbf{0})$, it does not contain any while loop, and it has only one recursive call in its body.*

A program whose computational instruction uses the method `decrement` cannot be safe as this method can only be given the type $\mathbf{BList}(\mathbf{0}) \rightarrow \mathbf{void}(\mathbf{0})$. Though it entails a lack of expressive power, changing the type system by allowing `decrement` to apply to tier $\mathbf{1}$ objects would allow codes like

```
while(!o.isEqual(l)){
    o.decrement();
},
```

where `isEqual` is a method testing the equality of two lists, and `l` is a \mathbf{BList} of Boolean numbers equal to 0. Clearly, such a loop can be executed exponentially in the size of the list `o`. This behavior is highly undesirable.

An important point to mention is that safety allows an easy way to perform expression subtyping for objects through the use of cloning. In other words, it is possible to bring a reference type expression from tier $\mathbf{1}$ to tier $\mathbf{0}$ based on the premise that it has been cloned in memory. This was already true for primitive data by rule (Ass). Thus a form of subtyping that does not break the non-interference properties is admissible as illustrated by the following example.

Example 3.1.18. *The method `clone`*

```
BList clone(){
    BList res0 := null ;
    int v0 := value1 ;
    if(queue1 == null){
        res0 := new BList(v0, null)0 ;
    }
```

```

    } else {
      res0 := new BList(v0, queue.clone()0);
    }
    return res;
  }

```

can be typed by $\text{BList}(\mathbf{1}) \rightarrow \text{BList}(\mathbf{0})$. Moreover, the method is safe, as there is only one recursive call and no while loop.

We are now ready to state the main characterization of polynomial time functions in terms of safe and terminating programs.

Theorem 3.1.7 ([HP18]). *The set of functions computed by safe and terminating programs is exactly FP.*

3.1.5 Type inference and declassification

The type systems of Figure 3.2, Figure 3.7, and Figure 3.9 have a decidable type inference.

Proposition 3.1.1 (Type inference). *Given a safe operator typing environment Δ and a program, deciding if there exists a variable typing environment Γ such that the program is safe using typing rules of Figure 3.2, Figure 3.7, or Figure 3.9 can be done in polynomial time in the size of the program.*

The proof of decidability of type system is provided in [HMP13] for processes and in [HP18] for OO programs. They rely on a reduction to 2-SAT which is known to be decidable in polynomial time. The result follows for the type system of Figure 3.2 as it is a strict subsystem of the system of Figure 3.7.

Observe that the decidability of type inference does not imply that the provided criteria (characterizing complexity classes) are decidable as they still require some extra hypothesis on program termination.

In [Mar11], the type system is based on a tier lattice allowing the type to perform some declassification mechanisms. Hence allowing programs of the shape

```

while(x > 0){
  x := x-1;
  y := y+1
};
while(y > 0){
  y := y-1;
  z := z+1
};

```

to type. However the above program cannot be typed using the type system of Figure 3.2. Indeed, the first loop enforces the y variable to be of tier $\mathbf{0}$, as the operator $+1$ is positive. The second loop enforces the y variable to be of tier $\mathbf{1}$ and, consequently, we obtain a contradiction.

However such kind of programs can still be analyzed in two ways.

- As suggested in [HMP13], the static analysis can be allowed to split programs in a constant number of subprograms and to perform the analysis on these subprograms. As a constant number of polynomial compositions remains polynomial, this would allow to analyze the above counter-example.

- Alternatively, declassification can be handled by allowing more than two tiers. In such a framework, the operator types would depend on the tier of their context as suggested by the program annotation below.

```

while(x2 > 0){
  x2 := x-1 ;
  y1 := y+1 : 1
};
while(y1 > 0){
  y1 := y-1 ;
  z:0 := z+1 : 0
};

```

Here the operator $+1$ would be restricted to have an output of tier strictly smaller than the tier of the outermost while loop. Hence it could be given the type $\mathbf{1} \rightarrow \mathbf{1}$ in the first while loop and would be enforced to be of type $\mathbf{0} \rightarrow \mathbf{0}$ in the second while loop.

3.2 Extensions of light/soft logics

On the light/soft logics side, a lot of work has been carried out on developing and studying the notion of proofs [BM10] in systems such as L^3 and L^4 . As already mentioned, proof-nets turn out to be the natural notion for studying reductions in the lambda calculus.

Extensions of this typing discipline have also been considered by several authors to capture new computational paradigms such as multi-threaded programs [MA11, Mad12], process calculi [DLMS10, DLMS16], and quantum programs [DLMZ10]. We will briefly describe them and discuss them in this section. This description is non-exhaustive but covers some of the most important aspects: extension to a concurrent programming language; handling of imperative features, threads, and side effects; and extension to a quantum programming language and characterizations of quantum polynomial time complexity classes.

3.2.1 Light logic and multi-threaded programs

Extension of light logics to programs with multi-threads and side effects have been considered by Amadio and Madet to capture elementary time and polynomial time in [MA11] and [Mad12], respectively.

Syntax and semantics of $\lambda^{!,\$,||}$

The language $\lambda^{!,\$,||}$ of [Mad12] guarantees termination in polynomial time, covering any scheduling of threads, and is defined by the following grammar:

Terms	$ \begin{aligned} M ::= & \mathbf{x} \mid r \mid * \mid \lambda x.M \mid M M \mid !M \mid \S M \\ & \text{let } !x = M \text{ in } M \mid \text{let } \S x = M \text{ in } M \mid \\ & \text{get}(r) \mid \text{set}(r, M) \mid (M \parallel M) \end{aligned} $
Stores	$\mathcal{S} ::= r \leftarrow M \mid (\mathcal{S} \parallel \mathcal{S})$
Program	$\mathcal{P} ::= r \leftarrow M \mid \mathcal{S} \mid (\mathcal{P} \parallel \mathcal{P}),$

where r belongs to a fixed set of regions (memory locations), $*$ is the unit, the operator $\text{get}(r)$ reads the region r , the operator $\text{set}(r, M)$ assigns M to the region r , $r \leftarrow M$. A store is a parallel composition of assignments of a term M to a region r , $r \leftarrow M$. A program is a parallel composition of terms and stores. Programs are always considered up to the

following structural equivalence rules $(\mathcal{P}_1 || \mathcal{P}_2) \equiv (\mathcal{P}_2 || \mathcal{P}_1)$ and $(\mathcal{P}_1 || (\mathcal{P}_2 || \mathcal{P}_3)) \equiv ((\mathcal{P}_1 || \mathcal{P}_2) || \mathcal{P}_3)$. The depth $d(\mathcal{P})$ of a program \mathcal{P} is the maximal number of nested modalities and its size $|\mathcal{P}|$ is the number of symbols in \mathcal{P} .

Reduction contexts are defined by the following grammar:

$$\mathcal{E} ::= [] \mid \mathcal{E} M \mid V \mathcal{E} \mid \S \mathcal{E} \mid \text{let } !x = \mathcal{E} \text{ in } M \mid \text{let } \S x = \mathcal{E} \text{ in } M \mid \text{set}(r, \mathcal{E}) \mid (\mathcal{E} || \mathcal{P}) \mid (\mathcal{P} || \mathcal{E}),$$

where V is a *value* defined by $V ::= x \mid r \mid * \mid \lambda x.M \mid \S V \mid !V$.

A Call-By-Value (CBV) evaluation strategy can then be defined:

- $\mathcal{E}[(\lambda x.M) V] \rightarrow_v \mathcal{E}[M\{V/x\}]$,
- $\mathcal{E}[\text{let } !x = !V \text{ in } M] \rightarrow_v \mathcal{E}[M\{V/x\}]$,
- $\mathcal{E}[\text{let } \S x = \S V \text{ in } M] \rightarrow_v \mathcal{E}[M\{V/x\}]$,
- $\mathcal{E}[\text{get}(r) || r \Leftarrow V] \rightarrow_v \mathcal{E}[V]$,
- $\mathcal{E}[\text{set}(r, V)] \rightarrow_v \mathcal{E}[*] || r \Leftarrow V$,
- $\mathcal{E}[* || M] \rightarrow_v \mathcal{E}[M]$.

Notice that, by definition of reduction contexts, no reduction occurs under a $!$.

Light type system for $\lambda^{!,\S,||}$

Before introducing the type system, define a *region context* R to be a mapping from some finite set of regions $\text{dom}(R)$ to a natural number, noted $R = r_1 : n_1, \dots, r_k : n_k$, for $n_i \in \mathbb{N}$ and $\text{dom}(R) = \{r_1, \dots, r_k\}$. Define a variable context Γ to be a mapping from some finite set of variables to a usage in $\{!, \S, \lambda\}$, noted $\Gamma = x_1 : u_1, \dots, x_k : u_k$, for $u_i \in \{!, \S, \lambda\}$ and $\text{dom}(\Gamma) = \{x_1, \dots, x_k\}$. The resource usage indicates a constraint on the variable bound. Let Γ_u , $u \in \{!, \S, \lambda\}$, be a shorthand notation for the variable context Γ where all variable in $\text{dom}(\Gamma)$ have usage u .

The light linear depth type system is presented in Figure 3.10. This type system works in a standard way through a stratification mechanism by depth level. It is worth mentioning that the type system of Figure 3.10 does not prevent programs from going wrong (deadlocks, ...). An improved version preventing such behaviors has also been presented in [Mad12].

Let us consider a small example.

Example 3.2.1. Let `add` be the standard encoding of addition over Church numerals of type $\text{Nat} \multimap \text{Nat} \multimap \text{Nat}$ with $\text{Nat} = \forall \alpha.!(\alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha)$. The term

$$\text{inc} = \lambda x. \lambda z. (\S(\text{set}(x, \text{let } !y = \text{get}(x) \text{ in } !\text{add}(\underline{1}, y))) || z)$$

increases the Church numeral stored at some location. Its well-formedness can be derived as

$$\begin{array}{c}
\frac{\mathbf{x} : \lambda \in \Gamma}{R; \Gamma \vdash^n \mathbf{x}} \text{ (Var)} \quad \frac{}{R; \Gamma \vdash^n * } \text{ (Unit)} \quad \frac{}{R; \Gamma \vdash^n r} \text{ (Reg)} \\
\\
\frac{FO(\mathbf{x}, M) = 1 \quad R; \Gamma, \mathbf{x} : \lambda \vdash^n M}{R; \Gamma \vdash^n \lambda \mathbf{x}. M} \text{ (Abs)} \quad \frac{R; \Gamma \vdash^n M \quad R; \Gamma \vdash^n N}{R; \Gamma \vdash^n M N} \text{ (App)} \\
\\
\frac{FO(M) \leq 1 \quad R; \Gamma_\lambda \vdash^{n+1} M}{R; \Gamma!, \Delta_\S, \Omega_\lambda \vdash^n !M} \text{ (Pr!)} \quad \frac{FO(\mathbf{x}, N) \geq 1 \quad R; \Gamma \vdash^n M \quad R; \Gamma, \mathbf{x} : ! \vdash^n N}{R; \Gamma \vdash^n \mathbf{let} \ !\mathbf{x} = M \ \mathbf{in} \ N} \text{ (E!)} \\
\\
\frac{FO(M) \leq 1 \quad R; \Gamma_\lambda, \Delta_\lambda \vdash^{n+1} M}{R; \Gamma!, \Delta_\S, \Omega_\lambda \vdash^n \S M} \text{ (Pr\S)} \quad \frac{FO(\mathbf{x}, N) = 1 \quad R; \Gamma \vdash^n M \quad R; \Gamma, \mathbf{x} : \S \vdash^n N}{R; \Gamma \vdash^n \mathbf{let} \ \S \mathbf{x} = M \ \mathbf{in} \ N} \text{ (E\S)} \\
\\
\frac{r : n \in R}{R; \Gamma \vdash^n \mathbf{get}(r)} \text{ (Get)} \quad \frac{r : n \in R \quad R; \Gamma \vdash^n M}{R; \Gamma \vdash^n \mathbf{set}(r, M)} \text{ (Set)} \\
\\
\frac{r : n \in R \quad R; \Gamma \vdash^n M}{R; \Gamma \vdash^0 r \Leftarrow M} \text{ (Str)} \quad \frac{R; \Gamma \vdash^n \mathcal{P}_1 \quad R; \Gamma \vdash^n \mathcal{P}_2}{R; \Gamma \vdash^n \mathcal{P}_1 || \mathcal{P}_2} \text{ (Par)}
\end{array}$$

Figure 3.10: LLL-based well-formedness rules for $\lambda^{!,\S,||}$

follows.

$$\begin{array}{c}
 \vdots \\
 \frac{}{\mathbf{x} : 1; \vdash^1 \text{get}(\mathbf{x})} \text{(Get)} \quad \frac{\mathbf{x} : 1; \mathbf{y} : \lambda \vdash^1 \text{add}(\underline{1}, \mathbf{y})}{\mathbf{x} : 1; \mathbf{y} : ! \vdash^1 \text{!add}(\underline{1}, \mathbf{y})} \text{(Pr}_{\eta}) \\
 \frac{}{\mathbf{x} : 1; \vdash^1 \text{let } !\mathbf{y} = \text{get}(\mathbf{x}) \text{ in } \text{!add}(\underline{1}, \mathbf{y})} \text{(E}_{\text{!}}) \\
 \frac{}{\mathbf{x} : 1; \vdash^1 \text{set}(\mathbf{x}, \text{let } !\mathbf{y} = \text{get}(\mathbf{x}) \text{ in } \text{!add}(\mathbf{y}, \mathbf{y}))} \text{(Set)} \\
 \frac{}{\mathbf{x} : 1; \mathbf{x} : \lambda, \mathbf{z} : \lambda \vdash^0 \S \text{set}(\mathbf{x}, \text{let } !\mathbf{y} = \text{get}(\mathbf{x}) \text{ in } \text{!add}(\underline{1}, \mathbf{y}))} \text{(Pr}_{\S}) \quad \frac{}{\mathbf{x} : 1; \mathbf{x} : \lambda, \mathbf{z} : \lambda \vdash^0 \mathbf{z}} \text{(Var)} \\
 \frac{}{\mathbf{x} : 1; \mathbf{x} : \lambda, \mathbf{z} : \lambda \vdash^0 \S (\text{set}(\mathbf{x}, \text{let } !\mathbf{y} = \text{get}(\mathbf{x}) \text{ in } \text{!add}(\underline{1}, \mathbf{y})) \parallel \mathbf{z})} \text{(Abs)} \quad \frac{}{\mathbf{x} : 1; \mathbf{x} : \lambda \vdash^0 \lambda \mathbf{z}. (\S (\text{set}(\mathbf{x}, \text{let } !\mathbf{y} = \text{get}(\mathbf{x}) \text{ in } \text{!add}(\underline{1}, \mathbf{y})) \parallel \mathbf{z}))} \text{(Par)} \\
 \frac{}{\mathbf{x} : 1; \vdash^0 \text{inc}} \text{(Abs)}
 \end{array}$$

The last two rules can be applied as the number of free occurrences of \mathbf{x} and \mathbf{z} in the term is exactly 1. The promotion rules Pr_{\S} and Pr_{η} can be applied for the same reason.

We can now state the main result, a polynomial time soundness result on such programs.

Theorem 3.2.1 ([Mad12]). *Given a program \mathcal{P} , if there are Γ , R , and n such that $R; \Gamma \vdash^n \mathcal{P}$ then the CBV evaluation of \mathcal{P} can be computed in time $O(|\mathcal{P}|^{2^{d(\mathcal{P})}})$.*

A polynomial completeness result also holds for FP as the type system extends LAL. An extension taking into account thread generation using higher-order references is also presented in [Mad12].

3.2.2 Soft logic and process calculi

An extension of SLL in the context of a higher-order process calculus $\text{HO}\pi$ has been studied by Dal Lago, Martini, and Sangiorgi in [DLMS10, DLMS16], where a soundness result is shown: typable processes terminate in polynomial time.

Syntax and semantics of $\text{LHO}\pi$

Considered processes are defined by the following grammar.

$$\begin{array}{ll}
 \text{Processes} & P ::= 0 \mid (P|P) \mid a(\mathbf{x}).P \mid \bar{a}\langle V \rangle.P \mid (\nu a)P|V \mid V \\
 \text{Values} & V ::= * \mid \mathbf{x} \mid \lambda \mathbf{x}.P
 \end{array}$$

Before introducing a linear type system, [DLMS16] extends standard processes into a language of linear processes $\text{LHO}\pi$ defined by the following grammar and reduction.

$$\begin{array}{ll}
 \text{Linear Processes} & LP ::= 0 \mid (LP|LP) \mid a(\mathbf{x}).LP \mid \bar{a}\langle LV \rangle.LP \mid (\nu a)LP \\
 & \quad \mid LV \mid LV \mid a(!\mathbf{x}).LP \\
 \text{Linear Values} & LV ::= * \mid \mathbf{x} \mid \lambda \mathbf{x}.LP \mid \lambda !\mathbf{x}.LP \mid !LV
 \end{array}$$

Processes in $\text{HO}\pi$ can be transformed into $\text{LHO}\pi$ processes using the following embedding $(-)^{\bullet}$.

$$\begin{array}{ll}
 (\bar{a}\langle V \rangle.P)^{\bullet} = \bar{a}\langle !\langle V \rangle^{\bullet} \rangle.(P)^{\bullet} & (\bar{a}(\mathbf{x}).P)^{\bullet} = a(!\mathbf{x}).(P)^{\bullet} \\
 (\lambda \mathbf{x}.P)^{\bullet} = \lambda !\mathbf{x}.(P)^{\bullet} & (V \ W)^{\bullet} = (V)^{\bullet}!\langle W \rangle^{\bullet} \\
 (P|Q)^{\bullet} = (P)^{\bullet}|(Q)^{\bullet} & ((\nu a)P)^{\bullet} = (\nu a)(P)^{\bullet} \\
 (k)^{\bullet} = k, \quad k \in \{0, *, \mathbf{x}\}. &
 \end{array}$$

The reduction relation \rightarrow on closed linear processes is standard and defined in Figure 3.11.

$$\begin{array}{c}
 \frac{}{a(\mathbf{x}).LP|\bar{a}\langle LV\rangle.LQ \rightarrow LP\{V/\mathbf{x}\}|LQ} \quad \frac{}{\lambda\mathbf{x}.LP LV \rightarrow LP\{V/\mathbf{x}\}} \\
 \frac{}{a(!\mathbf{x}).LP|\bar{a}\langle !LV\rangle.LQ \rightarrow P\{LV/\mathbf{x}\}|LQ} \quad \frac{}{\lambda!\mathbf{x}.LP !V \rightarrow LP\{LV/\mathbf{x}\}} \\
 \frac{LP \rightarrow LQ}{LR|LP \rightarrow LR|LQ} \quad \frac{LP \rightarrow QL}{(\nu a)LP \rightarrow (\nu a)LQ} \quad \frac{LR \equiv LP \quad LP \rightarrow LQ \quad LQ \equiv LS}{LR \rightarrow LS}
 \end{array}$$

Figure 3.11: Operational semantics of LHO π

\equiv is defined to be the smallest congruence closed under the rules:

$$\begin{array}{l}
 LP \equiv LQ, \text{ if } LP =_{\alpha} LQ, \\
 (LP|LQ) \equiv (LQ|LP) \\
 ((\nu a)LP|LQ) \equiv (\nu a)(LP|LQ), \text{ provided } a \notin FV(LQ). \\
 (LP|(LQ|LR)) \equiv ((LP|LQ)|LR), \\
 (\nu a)((\nu b)LP) \equiv (\nu b)((\nu a)LP),
 \end{array}$$

The standard rules of the π -calculus ($LP|0 \equiv LP$ and $(\nu a)0 \equiv 0$) are withdrawn from the congruence as they are not well-suited for a resource sensitive analysis.

Soft type system for LHO π

The soft type system of [DLMS10] is then introduced in Figure 3.12. As in previous section, a

$$\begin{array}{c}
 \frac{k \in \{*, 0\}}{\#\Gamma \vdash_{SP} k} \quad \frac{\Gamma, \#\Omega \vdash_{SP} LP \quad \Delta, \#\Omega \vdash_{SP} LQ}{\Gamma, \Delta, \#\Omega \vdash_{SP} LP|LQ} \quad \frac{\Gamma, \mathbf{x} : \lambda \vdash_{SP} LP}{\Gamma \vdash_{SP} \lambda\mathbf{x}.LP} \quad \frac{\Gamma \vdash_{SP} LP}{\Gamma \vdash_{SP} (\nu a)LP} \\
 \frac{\Gamma, \mathbf{x} : \lambda \vdash_{SP} LP}{\Gamma \vdash_{SP} a(\mathbf{x}).LP} \quad \frac{\Gamma, \mathbf{x} : u \vdash_{SP} LP \quad u \in \{!, \#\}}{\Gamma \vdash_{SP} a(!\mathbf{x}).LP} \\
 \frac{\Gamma, \mathbf{x} : u \vdash_{SP} LP \quad u \in \{!, \#\}}{\Gamma \vdash_{SP} \lambda!\mathbf{x}.LP} \quad \frac{u \in \{\lambda, \#\}}{\#\Gamma, \mathbf{x} : u \vdash_{SP} \mathbf{x}} \\
 \frac{\Gamma, \#\Omega \vdash_{SP} LV \quad \Delta, \#\Omega \vdash_{SP} LW}{\Gamma, \Delta, \#\Omega \vdash_{SP} LV LW} \quad \frac{\Gamma \vdash_{SP} LV}{!\Gamma, \#\Omega \vdash_{SP} !LV} \quad \frac{\Gamma, \#\Omega \vdash_{SP} LV \quad \Delta, \#\Omega \vdash_{SP} LP}{\Gamma, \Delta, \#\Omega \vdash_{SP} \bar{a}\langle LV\rangle.LP}
 \end{array}$$

Figure 3.12: SLL-based process type system

variable environment Γ is of the shape $\mathbf{x}_1 : u_1, \dots, \mathbf{x}_n : u_n$, where each variable \mathbf{x}_i comes with a unique usage $u_i \in \{\lambda, !, \#\}$, where $\#$ is a mark for contraction or weakening. For a given usage $u \in \{\lambda, !, \#\}$, $u\Gamma$ stands for an environment where all variables have usage u . In a context of the shape $\Gamma, u\Delta$, all variables in Γ are assumed to be of usage distinct from u .

As usual, the depth $d(P)$ of a process P in $\text{LHO}\pi$ is the maximal number of nested modalities and the size of a $|P|$ process P is its number of symbols.

Again, this system has a inherent stratification property ensuring that linear variables (of usage λ) occur exactly once at depth 0, non-linear variables of usage $!$ occur exactly once at depth 1, and non-linear variables of usage $\#$ occur at depth 0. This latter variables may occur several times.

Theorem 3.2.2 ([DLMS10]). *There are polynomials $(P_n)_{n \in \mathbb{N}}$ such that for each process P , if $\vdash_{\text{SP}} P$ and $P \rightarrow^m Q$ then $\max(m, |Q|) \leq P_{d(P)}(|P|)$.*

The property of Theorem 3.2.2 can be seen as restrictive for processes as, by essence, it forbids any non-terminating process from being analyzed. To overcome this issue, an extension to non-terminating processes has been studied in [DLMS16], where a polynomial bound is ensured on the number of internal computation steps performed by a process between any two interactions with its environment.

3.2.3 Soft linear logic and quantum programs

In [DLMZ10], the methodology of SLL is applied to a restricted quantum programming language by Dal Lago, Masini, and Zorzi. Measurement (see Chapter 5 for a discussion) is not handled by the language itself and only occurs at the end of a computation. [DLMZ10] introduces a quantum programming language SQ with a soft linear logic based type system that is shown to be sound and complete with respect to polynomial time Quantum Turing Machines (QTM) (see [Deu85] for a definition), hence providing characterizations of the three quantum complexity classes EQP, BQP, and ZQP (see [BV97] for a definition).

Syntax and semantics of SQ

We will not present the full quantum computation paradigm in this subsection and we invite the interested reader to read either [Sel04] or [SV06] for an interesting presentation of an imperative quantum language and a functional quantum language, respectively.

Terms of SQ are defined by the following grammar:

$$\begin{array}{ll} \text{Patterns } \phi & ::= x \mid !x \mid \langle x_1, \dots, x_n \rangle \\ \text{Terms } M & ::= 0 \mid 1 \mid x \mid r \mid !M \mid U \mid \text{new}(M) \mid \langle M, \dots, M \rangle \mid M M \mid \lambda \phi. M, \end{array}$$

where x, x_1, \dots, x_n are variables for classical data, r is a variable for quantum data, and U is a unitary operator on the 2^n dimensional Hilbert space $\mathcal{H}(\{0, 1\}^n)$. Recall that a unitary operator is a bounded linear operator $U : \mathcal{H} \rightarrow \mathcal{H}$ on a Hilbert space \mathcal{H} that satisfies $U^\dagger U = U U^\dagger = I$, where U^\dagger is the conjugate transpose of U and I is the identity operator on \mathcal{H} , *e.g.* the Hadamard gate is an example of such an operator.

Given a term M , let $Q(M)$ be the set of quantum variables occurring in M . A preconfiguration is a triplet $[Q, \mathcal{QV}, M]$ consisting in a set of quantum variables \mathcal{QV} such that $Q(M) \subseteq \mathcal{QV}$, a state Q of the Hilbert space $\mathcal{H}(\{0, 1\}^{\text{card}(\mathcal{QV})})$, and a term M . A configuration is just an equivalence class of preconfigurations obtained from a preconfiguration $[Q, \mathcal{QV}, M]$ and whose term and set of quantum variables can be obtained by a bijective renaming of quantum variables in \mathcal{QV} . The reduction rules consist in standard reduction rules on configuration described in Figure 3.13 together with commutation rules and contextual closure rules that we avoid to present here (the full rules are presented in [DLMZ10]). Let \rightarrow^* be the reflexive and transitive closure of the commutative and contextual closure of \rightarrow .

$$\begin{aligned}
 & [\mathcal{Q}, \mathcal{QV}, (\lambda x.M) \mathbb{N}] \rightarrow [\mathcal{Q}, \mathcal{QV}, M\{\mathbb{N}/x\}] \\
 & [\mathcal{Q}, \mathcal{QV}, (\lambda \langle x_1, \dots, x_n \rangle.M) \langle r_1, \dots, r_n \rangle] \rightarrow [\mathcal{Q}, \mathcal{QV}, M\{r_1/x_1, \dots, r_n/x_n\}] \\
 & [\mathcal{Q}, \mathcal{QV}, (\lambda !x.M) !\mathbb{N}] \rightarrow [\mathcal{Q}, \mathcal{QV}, M\{\mathbb{N}/x\}] \\
 & [\mathcal{Q}, \mathcal{QV}, U \langle r_1, \dots, r_n \rangle] \rightarrow [U_{r_1, \dots, r_n} \mathcal{Q}, \mathcal{QV}, \langle r_1, \dots, r_n \rangle] \\
 & [\mathcal{Q}, \mathcal{QV}, \mathbf{new}(i)] \rightarrow [\mathcal{Q} \otimes |r \mapsto i\rangle, \mathcal{QV} \cup \{r\}, r] \quad i \in \{0, 1\}
 \end{aligned}$$

Figure 3.13: Standard reduction rules of SQ

In the last rule of Figure 3.13, \otimes is the standard tensor product of Hilbert spaces and r is a fresh quantum variable. In the penultimate rule, U_{r_1, \dots, r_n} stands for the unitary operator transformation applied on qubits referenced by r_1, \dots, r_n in the state \mathcal{Q} . We do not present the full details here to avoid technicalities.

Soft type system for SQ

As in previous Subsection, a typing environment Γ comes with type annotations on its variables. The typing environments $!\Gamma$ and $\#\Gamma$ are obtained from Γ by annotating all variables with $!$ and $\#$, respectively. The type system of SQ is presented in Figure 3.14 where it is implicitly assumed that each classical or quantum variable occurs at most once in each environment of a typing judgment. The aim of the type system is again to enforce some kind of stratification discipline ensuring that a free variable x with no annotation occurs at most once in a typable term (and never under the $!$ modality), that a free variable x annotated by $\#$ occurs at least once in a typable term (and never under a $!$ modality), and that a free variable x annotated by $!$ occurs at most once in a typable term (it might occur under a $!$).

$$\begin{array}{c}
 \frac{k \in \{0, 1\}}{!\Gamma \vdash k} \quad \frac{}{!\Gamma, r \vdash r} \quad \frac{\dagger \in \{\emptyset, !, \#\}}{!\Gamma, \dagger x \vdash x} \quad \frac{; \forall i \leq n, \Gamma_i, \#\Delta_i \vdash M_i}{\cup_{i=1}^n \Gamma_i, \#\cup_{i=1}^n \Delta_i \vdash \langle M_1, \dots, M_n \rangle} \\
 \\
 \frac{\Gamma, \#\Delta \vdash M \quad \Gamma', \#\Delta' \vdash N}{\Gamma, \Gamma', \#\Delta, \#\Delta' \vdash M N} \quad \frac{\Gamma \vdash M}{!\Gamma, !\Delta \vdash !M} \quad \frac{\Gamma \vdash M}{\Gamma \vdash \mathbf{new}(M)} \\
 \\
 \frac{\Gamma, x_1, \dots, x_n \vdash M}{\Gamma \vdash \lambda \langle x_1, \dots, x_n \rangle.M} \quad \frac{\Gamma, x \vdash M}{\Gamma \vdash \lambda x.M} \quad \frac{\Gamma, \dagger x \vdash M \quad \dagger \in \{\#, !\}}{\Gamma \vdash \lambda !x.M}
 \end{array}$$

Figure 3.14: SQ type system

Before stating the main characterizations of [DLMZ10], we first show how to encode decision problems in the language.

For that purpose, we introduce some preliminary notations corresponding to data structures encoding presented in [DLMZ10]. Given some terms M_1, \dots, M_n , let $[M_1, \dots, M_n]$ be an encoding of the sequence of terms M_1, \dots, M_n . For a binary string t , let \tilde{t} be an encoding the binary string t as a typable term of SQ. The notation $!^n M$ stands for $! \dots !M$, n times.

A term M outputs the binary string $s \in \{0, 1\}^*$ with probability p on input N if there is a constant $m \geq |s|$ such that $[1, \emptyset, M N] \rightarrow^* [Q, \{r_1, \dots, r_n\}, [r_1, \dots, r_n]]$ and the probability of observing s when projecting Q into $\mathcal{H}(\{0, 1\}^{m-|s|})$ is exactly p .

Definition 3.2.1. *Given $n \in \mathbb{N}$, two binary strings $s, r \in \{0, 1\}^*$, and $p \in [0, 1]$, a typable term M of SQ and a language $L \subseteq \{0, 1\}^*$.*

- M (n, s, r, p) -decides L if and only if for every binary string t the following two conditions hold:
 - M outputs s with probability at least p on input $!^n \tilde{t}$, if $t \in L$,
 - M outputs r with probability at least p on input $!^n \tilde{t}$, if $t \notin L$.
- M is (n, s, r) -error-free if and only if for every binary string t , the following two conditions hold on input $!^n \tilde{t}$:
 - if M outputs s with positive probability then M outputs r with null probability,
 - if M outputs r with positive probability then M outputs s with null probability.

Definition 3.2.2. *The class of language ESQ , BSQ and ZSQ are defined as follows.*

- ESQ is the class of languages that are $(n, s, r, 1)$ -decided by a term of SQ .
- BSQ is the class of languages that are (n, s, r, p) -decided by a term of SQ , with $p > 1/2$.
- ZSQ is the class of languages that are (n, s, r, p) -decided by a (n, s, r) -error-free term of SQ , with $p > 1/2$.

We recall briefly the definition of the quantum complexity classes EQP , BQP , and ZQP of [BV97].

Definition 3.2.3. *The complexity classes EQP , BQP , and ZQP are defined as follows.*

- EQP is the class of decision problems computed by QTM s that output the correct answer with probability 1 and run in polynomial time.
- BQP is the class of decision problems computed by QTM s that output the correct answer with probability $p > 1/2$ and run in polynomial time.
- ZQP is the class of decision problems computed by QTM s that output the answer with probability $p > 1/2$ and error of probability 0, and run in polynomial time.

Now we are ready to state the characterization of these complexity classes.

Theorem 3.2.3 ([DLMZ10]). $\forall x \in \{E, B, Z\}, xSQ = xQP$.

3.2.4 Miscellaneous

In this subsection, we briefly discuss some other extensions or alternative uses of soft/light logics approaches in the framework of proof-nets, interaction-nets, and model theory.

Proof-nets and interaction nets

The paper [Per18] introduces a decidable syntactic proof-net-based subsystem of linear logic, called SDNLL, that is sound and complete for polynomial time and that strictly embeds LLL. Although not directly related to light logics, the complexity of functional programs has been studied in [GM16] using an interaction-net computational model, a generalization of linear logic proof-nets. This model has been combined with sized types to certify time and space complexity bounds for both sequential and parallel proof-net reductions.

Categorical and realizability models

Categorical models for ELL and SLL have been provided in [LTDF06]. The paper [Red07] has provided an axiomatization of categorical-based SLL models. Categorical models for ELL and LLL have been studied in [Bai04a]. Realizability models have been adapted to the ICC setting in [DLH05, DLH11] for soundness proofs of EAL and Soft Affine Logic (SAL). They have also been adapted to show the soundness of a fragment of second order linear logic with type fixpoints (with respect to FP) in [BT10] and of LAL in [BM12].

3.3 Extensions of interpretations

As presented in Section 2.2 and Section 2.3, the interpretation method and the quasi-interpretation method were originally designed to study termination properties and complexity properties, respectively, of first order TRSs. In this section, we discuss their extensions to other programming paradigms including higher-order TRSs [BMP07, BDL12, BDL16], first order and higher-order functional programs [GP09a, GP09b, GP15b, HP17], Object Oriented programs [MP08a], Java bytecode [ACGDZJ04], cooperative threads [ADZ04], and polygraphs [BG08, BG07].

3.3.1 Higher-order rewrite systems

In this subsection, we focus on extensions of interpretation methods to higher-order rewriting. The considered computational model will be Simply Typed TRSs (STTRSs) studied by Yamada [Yam01]. They consist in a simple extension of TRS with higher-order functions. Consider a fixed set of basic datatypes \mathcal{B} . The set of types is defined by:

$$A ::= b \mid A_1 \times \dots \times A_n \rightarrow A,$$

with $b \in \mathcal{B}$.

The set of terms is defined by:

$$t^A ::= \mathbf{x}^A \mid \mathbf{c}^{b_1 \times \dots \times b_n \rightarrow b} \mid \mathbf{fun}^A \mid (t^{A_1 \times \dots \times A_n \rightarrow A} t_1^{A_1} \dots t_n^{A_n})^A,$$

where \mathbf{x} is a variable in \mathcal{X} , \mathbf{c} is a constructor symbol in \mathcal{C} , and \mathbf{fun} is a function symbol in \mathcal{F} . Constructor symbol types are restricted to functionals on basic types. As usual, for a given term t , $\mathbb{V}(t)$ denotes the set of variables in t . Consequently, $\mathbb{V}(t) \subseteq \mathcal{X}$. A pattern p is a term with no occurrence of a function symbol.

A STTRS consists in a set of non-overlapping rules $l^A \rightarrow r^A$, satisfying:

- $\mathbb{V}(r) \subseteq \mathbb{V}(l)$ and variables occur once in l ,
- $l = \mathbf{fun}^{A_1 \times \dots \times A_n \rightarrow A} p_1^{A_1} \dots p_n^{A_n}$, for some patterns p_1, \dots, p_n .

The underlying CBV rewrite relation is defined in a standard way by verifying that types match in a substitution. A substitution σ maps variables to values of the same type that are defined inductively by

$$v ::= c^{b_1 \times \dots \times b_n \rightarrow b} v_1 \dots v_n \mid \text{fun}^{A_1 \times \dots \times A_n \rightarrow A} v_1 \dots v_k,$$

with $k < n$. Contexts are defined in a standard manner. We write $t \rightarrow_{\mathcal{R}} s$ if there are a context $C[\diamond^A]$, a rule $l^A \rightarrow r^A$, and a substitution σ such that $t = C[l\sigma]$ and $s = C[r\sigma]$.

Example 3.3.1. Consider the following STTRS as an illustrating example:

$$\begin{aligned} (\text{fold } f \ z \ \text{nil}) &\rightarrow z, \\ (\text{fold } f \ z \ (c \ \text{hd} \ \text{tl})) &\rightarrow (f \ \text{hd} \ (\text{fold } f \ z \ \text{tl})), \\ (\text{add } 0 \ y) &\rightarrow y, \\ (\text{add } (x+1) \ y) &\rightarrow (\text{add } x \ y) + 1, \\ (\text{sum } 1) &\rightarrow (\text{fold } \text{add } 0 \ 1), \end{aligned}$$

with $\mathcal{X} = \{x^{\text{Nat}}, y^{\text{Nat}}, z^{\text{Nat}}, f^{\text{Nat} \times \text{Nat} \rightarrow \text{Nat}}, \text{hd}^{\text{Nat}}, \text{tl}^{L(\text{Nat})}, 1^{L(\text{Nat})}\}$, $\mathcal{C} = \{0^{\text{Nat}}, +1^{\text{Nat} \rightarrow \text{Nat}}, \text{nil}^{L(\text{Nat})}, c^{\text{Nat} \times L(\text{Nat}) \rightarrow L(\text{Nat})}\}$, and $\mathcal{F} = \{\text{fold}^{(\text{Nat} \times \text{Nat} \rightarrow \text{Nat}) \times \text{Nat} \times L(\text{Nat}) \rightarrow \text{Nat}}, \text{add}^{\text{Nat} \times \text{Nat} \rightarrow \text{Nat}}, \text{sum}^{L(\text{Nat}) \rightarrow \text{Nat}}\}$, Nat being the basic type of unary numbers and $L(\text{Nat})$ being the basic type of lists of unary numbers.

Defunctionalization

STTRSs are handled in [BMP07] using defunctionalization techniques [Rey72, CD96]. See [Dan06] for an introduction to defunctionalization and Continuation-Passing Style (CPS). To avoid technical details, we just illustrate this transformation through the following example.

Example 3.3.2. The following TRS is the CPS defunctionalization of the STTRS of Example 3.3.1:

$$\begin{aligned} \overline{\text{fold}}(z, \text{nil}) &\rightarrow z, \\ \overline{\text{fold}}(z, c(\text{hd}, \text{tl})) &\rightarrow \text{app}(c_0(\text{hd}), \overline{\text{fold}}(z, \text{tl})), \\ \text{add}(0, y) &\rightarrow y, \\ \text{add}((x+1), y) &\rightarrow \text{add}(x, y) + 1, \\ \text{sum}(1) &\rightarrow \overline{\text{fold}}(0, 1), \\ \text{app}(c_0(x_1), x_2) &\rightarrow \text{add}(x_1, x_2). \end{aligned}$$

Let $DF(S)$ be the set of function symbols of a STTRS of type $b_1 \times \dots \times b_n \rightarrow b$, for some basic types b_1, \dots, b_n, b , and whose CPS defunctionalization is in S , for some set of TRSs S . We obtain the following straightforward characterizations.

Theorem 3.3.1 ([BMP07]). For $\mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+, \mathbb{R}^+\}$, we have:

- $\llbracket DF(QI_{\text{add}}^{\text{poly}}(\mathbb{K}, \geq_{\mathbb{K}}) \cap RPO\{p\}) \rrbracket = \text{FP}$,
- $\llbracket DF(QI_{\text{add}}^{\text{poly}}(\mathbb{K}, \geq_{\mathbb{K}}) \cap RPO\{l\}) \rrbracket = \text{FPSPACE}$.

Example 3.3.3. The TRS of Example 3.3.2 admits the following interpretation:

$$\begin{aligned}
 [0]_{\mathbb{K}} &= [\mathbf{nil}]_{\mathbb{K}} = 0, \\
 [c_0]_{\mathbb{K}}(X) &= [+1]_{\mathbb{K}}(X) = X + 1, \\
 [c]_{\mathbb{K}}(X, Y) &= X + Y + 1, \\
 [\mathbf{fold}]_{\mathbb{K}}(X, Y) &= [\mathbf{add}]_{\mathbb{K}}(X, Y) = [\mathbf{app}]_{\mathbb{K}}(X, Y) = X + Y, \\
 [\mathbf{sum}]_{\mathbb{K}}(X) &= X.
 \end{aligned}$$

Moreover it can be oriented by RPO with status p . Consequently, the function symbol \mathbf{sum} of the STTRS of Example 3.3.1 computes a function in FP.

Some modular extensions using the notion of stratified union of Section 2.3.5 are also studied in [BMP07].

Related applied works studying the complexity of higher-order functionals can be found in [ADLM15]. The programs are analyzed automatically by applying program transformations to a defunctionalization and using standard complexity analysis techniques for first order TRSs. Similar clock based transformations were also studied in [DLR15]. More generally, techniques for reasoning formally about execution costs of higher-order programs were studied in [San90, San91].

Higher-order interpretations

The direct study of STTRS complexity is tackled in [BDL12, BDL16] by extending interpretations to higher-order polynomials defined as the β -normal forms of terms generated by the following grammar:

$$M ::= \mathbf{x}^A \mid \mathbf{op} \mid (M^{A \rightarrow B} N^A)^B \mid (\lambda \mathbf{x}^A. M^B)^{A \rightarrow B},$$

with $\mathbf{op} \in \{+ : \mathit{Nat} \times \mathit{Nat} \rightarrow \mathit{Nat}, * : \mathit{Nat} \times \mathit{Nat} \rightarrow \mathit{Nat}\} \cup \{\underline{n} : \mathit{Nat} \mid \forall n \in \mathit{Nat}\}$ and

$$A, B ::= \mathit{Nat} \mid A \times B \mid A \rightarrow B.$$

Higher-Order Polynomials (HOP) are restricted to strictly increasing total functions relatively to a pointwise order such that $f >_{A \rightarrow B} g$ if and only if $\forall a \in A, f(a) >_B g(a)$.

STTRS types are interpreted inductively by $[\mathit{Nat}]_{\mathbb{N}} = \mathbb{N}$ and $[A_1 \times \dots \times A_n \rightarrow A]_{\mathbb{N}} = [A_1]_{\mathbb{N}} \times \dots \times [A_n]_{\mathbb{N}} \rightarrow [A]_{\mathbb{N}}$. An assignment of a STTRS is a map from symbols in $\mathcal{C} \cup \mathcal{F}$ to HOPs: each symbol $\mathbf{b}^A \in \mathcal{C} \cup \mathcal{F}$ is mapped to a HOP $[\mathbf{b}]_{\mathbb{N}}$ of type $[A]_{\mathbb{N}}$. Assignments are extended to terms in a standard way. Each variable \mathbf{x}^A of \mathcal{X} is interpreted by a variable $[\mathbf{x}]_{\mathbb{N}}$ of type $[A]_{\mathbb{N}}$. Each term of the shape $(\mathbf{b}^{A_1 \times \dots \times A_n \rightarrow A} t_1^{A_1} \dots t_n^{A_n})^A$ is mapped to $([\mathbf{b}]_{\mathbb{N}}^{[A_1]_{\mathbb{N}} \times \dots \times [A_n]_{\mathbb{N}} \rightarrow [A]_{\mathbb{N}}} [t_1]_{\mathbb{N}}^{[A_1]_{\mathbb{N}}} \dots [t_n]_{\mathbb{N}}^{[A_n]_{\mathbb{N}}})^{[A]_{\mathbb{N}}} \equiv_{\beta} M^{[A]_{\mathbb{N}}}$, for some HOP M .²⁵

Definition 3.3.1. An assignment $[-]_{\mathbb{N}}$ is a higher-order polynomial interpretation of a STTRS if for each rule $l^A \rightarrow r^A$, $[l]_{\mathbb{N}}^{[A]_{\mathbb{N}}} >_{[A]_{\mathbb{N}}} [r]_{\mathbb{N}}^{[A]_{\mathbb{N}}}$. If the interpretation of each symbol in \mathcal{C} is additive (see Definition 2.2.5) then $[-]_{\mathbb{N}}$ is an additive higher-order polynomial interpretation.

Let HOP_{add}^{poly} be the set of TRSs that admit an additive higher-order polynomial interpretation over \mathbb{N} . We obtain a characterization of FP extending the characterization of Theorem 2.2.2 to STTRS.

²⁵Some β reduction steps are needed to compute the β normal form M . Hence we consider HOPs up to β equivalence, \equiv_{β} .

Theorem 3.3.2 ([BDL12]). $\llbracket HOPI_{add}^{poly} \rrbracket = \text{FP}$.²⁶

Example 3.3.4 ([BDL12]). Consider the following STTRS:

$$\begin{aligned} (\text{map } f \text{ nil}) &\rightarrow \text{nil}, \\ (\text{map } f \text{ (c hd tl)}) &\rightarrow (\text{c (f hd) (map } f \text{ tl)}). \end{aligned}$$

It admits the following additive higher-order polynomial interpretation:

$$\begin{aligned} [\text{nil}]_{\mathbb{N}}^{\mathbb{N}} &= 2, \\ [\text{c}]_{\mathbb{N}}^{\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}} &= \lambda n. \lambda m. m + n + 1, \\ [\text{map}]_{\mathbb{N}}^{(\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N} \rightarrow \mathbb{N}} &= \lambda \phi. \lambda n. n \times \phi(n). \end{aligned}$$

For the first rule, we check that:

$$\begin{aligned} [\text{map } f \text{ nil}]_{\mathbb{N}} &= \lambda \phi. \lambda n. n \times \phi(n) [\text{f}]_{\mathbb{N}} [\text{nil}]_{\mathbb{N}} \\ &= \lambda \phi. \lambda n. n \times \phi(n) [\text{f}]_{\mathbb{N}} 2 \\ &\equiv_{\beta} 2 \times [\text{f}]_{\mathbb{N}}(2) \\ &> 2 \\ &= [\text{nil}]_{\mathbb{N}}. \end{aligned}$$

The strict inequality holds as HOP are restricted to strictly increasing total functions and, consequently, $\forall n \in \mathbb{N}$, $[\text{f}]_{\mathbb{N}}(n) \geq n$. This explains the reason why the interpretation of `nil` has been set to 2.

We let the reader check that the strict inequality also holds for the second rule. For any first order function symbol `fun` (of the right type) admitting a polynomial interpretation, the function symbol `mapfun` defined by the rule $(\text{map}_{\text{fun}} \text{ l}) = (\text{map } \text{fun } \text{ l})$ admits an additive higher-order polynomial interpretation defined by: $[\text{map}_{\text{fun}}]_{\mathbb{N}} = \lambda l. ([\text{map}]_{\mathbb{N}} [\text{fun}]_{\mathbb{N}} l) + 1$, and, consequently, computes a function in FP, by Theorem 3.3.2.

As polynomial interpretations, the notion of higher-order polynomial interpretation suffers from a weak expressive power. It has been extended in [BDL16] to obtain a notion of higher-order quasi-interpretation. This latter notion has been combined to a syntactic termination criterion using a linearity restriction to provide a more expressive characterization of FP in [BDL16].

3.3.2 Functional programs

The issue of knowing whether interpretation methods and quasi-interpretation methods can be adapted to functional programs was open for a while: the difficulty was hidden in the basic nature of interpretations that provides a well-founded decreasing order on program reductions. While the decrease is explicit in TRSs as data (constructor symbols) are explicitly consumed by the rules in recursive calls, this phenomenon is less obvious for general functional programs where data may be consumed using destructors or pattern matching. This issue was tackled in two steps with an extension of interpretations to first order programs in [GP09a, GP09b, GP15b] and to higher-order programs in [HP17]. We will present this latter approach in this subsection.

Consider the following higher-order programming language:

$$M, N ::= x \mid c \mid \text{op} \mid M N \mid \lambda x. M \mid \text{letRec } f = M \mid \text{case } M \text{ of } c_1 \ x_1 : M_1 \mid \dots \mid c_n \ x_n : M_n,$$

²⁶This results holds for type 1 function symbols over basic types, e.g. unary numbers or binary words.

Let $(\mathbb{N}, \leq, \sqcup, \sqcap)$ be the set of natural numbers equipped with the usual ordering \leq , a max operator \sqcup and min operator \sqcap and let $\overline{\mathbb{N}}$ be $\mathbb{N} \cup \{\top\}$, where $\forall n \in \mathbb{N}, n \leq \top, n \sqcup \top = \top \sqcup n = \top$ and $n \sqcap \top = \top \sqcap n = n$. The strict order relation over natural numbers $<$ will also be used in the sequel and is extended in a somewhat unusual manner, by $\top < \top$.

The (higher-order) *interpretation* of a type is defined inductively by:

$$\begin{aligned} [\mathbf{b}] &= \overline{\mathbb{N}}, \text{ if } \mathbf{b} \text{ is a basic type,} \\ [\mathbf{T} \longrightarrow \mathbf{T}'] &= [\mathbf{T}] \longrightarrow^\uparrow [\mathbf{T}'], \text{ otherwise,} \end{aligned}$$

where $[\mathbf{T}] \longrightarrow^\uparrow [\mathbf{T}']$ denotes the set of total strictly monotonic functions from $[\mathbf{T}]$ to $[\mathbf{T}']$. As in previous section, a function F from the set A to the set B is strictly monotonic if and only if for each $X, Y \in A$, $X <_A Y$ implies $F(X) <_B F(Y)$, where $<_A$ is the usual ordering induced by $<$ and defined inductively by:

$$\begin{aligned} n <_{\overline{\mathbb{N}}} m, & \text{ if and only if } n < m, \\ F <_{A \longrightarrow^\uparrow B} G, & \text{ if and only if } \forall X \in A, F(X) <_B G(X). \end{aligned}$$

Example 3.3.6. *The type $\mathbf{T} = (\text{Nat} \longrightarrow \text{Nat}) \longrightarrow L(\text{Nat}) \longrightarrow L(\text{Nat})$ of the term $\text{letRec } \mathbf{f} = \mathbf{M}$ in Example 3.3.5 is interpreted by:*

$$[\mathbf{T}] = (\overline{\mathbb{N}} \longrightarrow^\uparrow \overline{\mathbb{N}}) \longrightarrow^\uparrow (\overline{\mathbb{N}} \longrightarrow^\uparrow \overline{\mathbb{N}}).$$

Each closed term of type \mathbf{T} will be interpreted by a functional in $[\mathbf{T}]$. The application is denoted as usual whereas we use the notation Λ for abstraction on this function space in order to avoid confusion between terms of our calculus and objects of the interpretation domain. Variables of the interpretation domain will be denoted using upper case letters. Moreover, we will sometimes use Church typing discipline in order to highlight the type of the bound variable in a lambda abstraction.

An important distinction between the terms of our language and the objects of the interpretation domain lies in the fact that beta-reduction is considered as an equivalence relation on (closed terms of) the interpretation domain, *i.e.* $(\Lambda X.F)G = F\{G/X\}$ underlying that $(\Lambda X.F)G$ and $F\{G/X\}$ are distinct notations that represent the same higher-order function. The same property holds for η -reduction, *i.e.* $\Lambda X.(FX)$ and F denote the same function.

Since we are interested in complete lattices, we need to complete each type $[\mathbf{T}]$ by a lower bound $\perp_{[\mathbf{T}]}$ and an upper bound $\top_{[\mathbf{T}]}$ inductively as follows:

$$\begin{aligned} \perp_{\overline{\mathbb{N}}} &= 0, & \top_{\overline{\mathbb{N}}} &= \top, \\ \perp_{[\mathbf{T} \longrightarrow \mathbf{T}']} &= \Lambda X^{[\mathbf{T}]}.\perp_{[\mathbf{T}']}, & \top_{[\mathbf{T} \longrightarrow \mathbf{T}']} &= \Lambda X^{[\mathbf{T}]}.\top_{[\mathbf{T}']}. \end{aligned}$$

Now we need to define a unit (or constant) cost function for any interpretation of type \mathbf{T} in order to take the cost of recursive calls into account. For that purpose, let $+$ denote natural number addition extended to $\overline{\mathbb{N}}$ by $\forall n, \top + n = n + \top = \top$. For each type $[\mathbf{T}]$, we define inductively a dyadic sum function $\oplus_{[\mathbf{T}]}$ by:

$$\begin{aligned} X^{\overline{\mathbb{N}}} \oplus_{\overline{\mathbb{N}}} Y^{\overline{\mathbb{N}}} &= X + Y, \\ F \oplus_{[\mathbf{T} \longrightarrow \mathbf{T}']} G &= \Lambda X^{[\mathbf{T}]}.(F(X) \oplus_{[\mathbf{T}']} G(X)). \end{aligned}$$

Let us also define the constant function $n_{[\mathbf{T}]}$, for each type \mathbf{T} and each integer $n \geq 1$, by:

$$\begin{aligned} n_{\overline{\mathbb{N}}} &= n, \\ n_{[\mathbf{T} \longrightarrow \mathbf{T}']} &= \Lambda X^{[\mathbf{T}]}.\perp_{[\mathbf{T}']}. \end{aligned}$$

- $[\mathbf{f}]_\rho = \rho(\mathbf{f})$, if $\mathbf{f} \in \mathbb{V}$,
 - $[\mathbf{c}]_\rho = 1 \oplus (\Lambda X_1 \dots \Lambda X_n \cdot \sum_{i=1}^n X_i)$, if $ar(\mathbf{c}) = n$,
 - $[\mathbf{MN}]_\rho = [\mathbf{M}]_\rho [\mathbf{N}]_\rho$,
 - $[\lambda \mathbf{x}.\mathbf{M}]_\rho = 1 \oplus (\Lambda [\mathbf{x}]_\rho \cdot [\mathbf{M}]_\rho)$,
 - $[\mathbf{case M of c}_1 \mathbf{x}_1 : \mathbf{M}_1 \dots | \mathbf{c}_n \mathbf{x}_n : \mathbf{M}_n]_\rho = 1 \oplus \sqcup_{1 \leq i \leq m} \{ [\mathbf{M}_i]_\rho \{ R_i / [\mathbf{x}_i]_\rho \} \mid \forall R_i \text{ s.t. } [\mathbf{M}]_\rho \geq [\mathbf{c}_i]_\rho R_i \}$,
 - $[\mathbf{letRec f = M}]_\rho = \sqcap \{ F \in [\mathbf{T}] \mid F \geq \Lambda [\mathbf{f}]_\rho \cdot [\mathbf{M}]_\rho (1 \oplus F) \}$.
-

Figure 3.16: Higher-order interpretation of a term

Once again, we will omit the type when it is unambiguous using the notation $n \oplus$ to denote the function $n_{[\mathbf{T}] \oplus [\mathbf{T}]}$ when $[\mathbf{T}]$ is clear from the typing context.

In the same spirit, we extend inductively the *max* and *min* operators \sqcup and \sqcap over $\bar{\mathbb{N}}$ to arbitrary higher-order functions F, G of type $[\mathbf{T}] \rightarrow^\uparrow [\mathbf{T}']$ by:

$$\begin{aligned} \sqcup^{[\mathbf{T}] \rightarrow^\uparrow [\mathbf{T}']} (F, G) &= \Lambda X^{[\mathbf{T}]} \cdot \sqcup^{[\mathbf{T}']} (F(X), G(X)), \\ \sqcap^{[\mathbf{T}] \rightarrow^\uparrow [\mathbf{T}']} (F, G) &= \Lambda X^{[\mathbf{T}]} \cdot \sqcap^{[\mathbf{T}']} (F(X), G(X)). \end{aligned}$$

In the following, we use the notations \perp , \top , \leq , $<$, \sqcup and \sqcap instead of $\perp_{[\mathbf{T}]}$, $\top_{[\mathbf{T}]}$, $\leq_{[\mathbf{T}]}$, $<_{[\mathbf{T}]}$, $\sqcup^{[\mathbf{T}]}$ and $\sqcap^{[\mathbf{T}]}$, respectively, when $[\mathbf{T}]$ is clear from the typing context.

Definition 3.3.2. A variable assignment, denoted ρ , is a map associating to each $\mathbf{f} \in \mathbb{V}$ of type \mathbf{T} , a variable F of type $[\mathbf{T}]$.

Definition 3.3.3 (Interpretation). Given a variable assignment ρ , an interpretation is the extension of ρ to well-typed terms, mapping each term of type \mathbf{T} to an object in $[\mathbf{T}]$ and defined in Figure 3.16 and extended to operators in such a way that $[\mathbf{op}]_\rho$ is a sup-interpretation, i.e. a total function such that:

$$\forall \mathbf{M}_1, \dots, \forall \mathbf{M}_n, [\mathbf{op M}_1 \dots \mathbf{M}_n]_\rho \geq [[\mathbf{op}](\mathbf{M}_1, \dots, \mathbf{M}_n)]_\rho.$$

As operator sup-interpretations are fixed, an interpretation should also be indexed by some mapping m assigning a sup-interpretation to each operator of the language. To simplify the formalism, we have omitted this mapping.

Let the size $|\mathbf{M}|$ of a term \mathbf{M} be defined by $|\mathbf{x}| = 0$, $|\mathbf{c}| = 1$, $|\mathbf{M N}| = |\mathbf{M}| + |\mathbf{N}|$, $|\lambda \mathbf{x}.\mathbf{M}| = 1 + |\mathbf{M}|$, $|\mathbf{letRec f = M}| = 1 + |\mathbf{M}|$, and $|\mathbf{case M of c}_1 \mathbf{x}_1 : \mathbf{M}_1 \dots | \mathbf{c}_n \mathbf{x}_n : \mathbf{M}_n| = 1 + |\mathbf{M}| + \sum_{i=1}^n |\mathbf{M}_i|$. First order values V are defined as in Subsection 2.2.1.

The developed notion of interpretation has the following properties.

Lemma 3.3.1 ([HP17]). For each value V , $[V]_\rho = |V|$.

Lemma 3.3.1 corresponds to Lemma 2.2.1 of Subsection 2.2.2. The equality is obtained as the additive constant of the interpretation of constructors is enforced to be 1 in Figure 3.16.

Lemma 3.3.2 ([HP17]). *For any program M , if $M \Rightarrow N$ then $[M]_\rho \geq [N]_\rho$. If $M \rightarrow_\alpha N$, $\alpha \in \{\beta, \text{letRec}, \text{case}\}$, then $[M]_\rho > [N]_\rho$.*

Lemma 3.3.2 is the higher-order counter-part of Lemma 2.3.1 of Subsection 2.3.2. Indeed, if M reduces to some value V then by transitivity, we obtain $[M]_\rho \geq [V]_\rho = |V|$, by Lemma 3.3.1.

Corollary 3.3.1 ([HP17]). *For any programs, M, N , such that $\emptyset; \Delta \vdash M N :: b$, with $b \in \mathbf{B}$, if $[M N]_\rho \neq \top$ then $M N$ terminates in a number of reduction steps in $O([M N]_\rho)$.*

Corollary 3.3.1 is the higher-order counter-part of Theorem 2.2.1 of Subsection 2.2.2. In the particular case where for any program N of the right type, $[M N]_\rho \neq \top$, the higher-order program M terminates on any input (hence, computes a total function).

We end the presentation of these interpretations by exhibiting the interpretation of the program of Example 3.3.5. We apply some relaxations on the interpretation (upper bounds) to simplify the presentation.

Example 3.3.7 ([HP17]). *Consider the program M of Example 3.3.5:*

$$\begin{aligned} \text{letRec } f &= \lambda g. \lambda x. \text{case } x \text{ of } c \ y \ z : c \ (g \ y) \ (f \ g \ z), \\ & \quad | \text{nil} : \text{nil}. \end{aligned}$$

By applying the rules of Figure 3.16, the following inequalities can be derived.

$$\begin{aligned} [M]_\rho &= \sqcap \{F \in [\mathbf{T}] \mid F \geq \Lambda[f]_\rho. [\lambda g. \lambda x. \text{case } x \text{ of } c \ y \ z : c \ (g \ y) \ (f \ g \ z) \mid \text{nil} : \text{nil}]_\rho (1 \oplus F)\} \\ &= \sqcap \{F \in [\mathbf{T}] \mid F \geq 2 \oplus (\Lambda[f]_\rho. \Lambda[g]_\rho. \Lambda[x]_\rho. [\text{case } x \text{ of } c \ y \ z : c \ (g \ y) \ (f \ g \ z) \mid \text{nil} : \text{nil}]_\rho (1 \oplus F))\} \\ &= \sqcap \{F \in [\mathbf{T}] \mid F \geq 3 \oplus (\Lambda[f]_\rho. \Lambda[g]_\rho. \Lambda[x]_\rho. \sqcup ([\text{nil}]_\rho, \sqcup_{[x]_\rho \geq [c(y,z)]_\rho} ([c(g \ y)(f \ g \ z)]_\rho)) (1 \oplus F))\} \\ &= \sqcap \{F \in [\mathbf{T}] \mid F \geq 5 \oplus (\Lambda[g]_\rho. \Lambda[x]_\rho. \sqcup_{[x]_\rho \geq 1 \oplus [y]_\rho + [z]_\rho} (([g]_\rho [y]_\rho) + (F [g]_\rho [z]_\rho)))\} \\ &\leq \sqcap \{F \in [\mathbf{T}] \mid F \geq 5 \oplus (\Lambda[g]_\rho. \Lambda[x]_\rho. (([g]_\rho ([x]_\rho - 1)) + (F [g]_\rho ([x]_\rho - 1))))\} \\ &\leq \Lambda[g]_\rho. \Lambda[x]_\rho. (5 \oplus ([g]_\rho [x]_\rho)) \times [x]_\rho \end{aligned}$$

In the penultimate line, we obtain an upper-bound on the interpretation by approximating the case interpretation, substituting $[x]_\rho - 1$ to both $[y]_\rho$ and $[z]_\rho$. In the last line, we obtain an upper-bound on the interpretation by approximating the **letRec** interpretation, just checking that the function $\Lambda[g]_\rho. \Lambda[x]_\rho. (5 \oplus ([g]_\rho [x]_\rho)) \times [x]_\rho$, where \times is the usual multiplication symbol over natural numbers, satisfies the inequality $F \geq 5 \oplus (\Lambda[g]_\rho. \Lambda[x]_\rho. (([g]_\rho ([x]_\rho - 1)) + (F [g]_\rho ([x]_\rho - 1))))$.

Consequently, the program M admits an interpretation bounded by $\Lambda[g]_\rho. \Lambda[x]_\rho. (5 \oplus ([g]_\rho [x]_\rho)) \times [x]_\rho$.

The properties of higher-order interpretations can be used to characterize FP and also higher-order polynomial time complexity classes. For that purpose, we need to restrict the space of interpretations to higher-order polynomials. These results will be presented in Section 4.2.

3.3.3 Object oriented programs

Sup-interpretations have also been extended to the case of OO programming in [MP08a].

The syntax of considered OO programs is very close to the syntax of programs in Subsection 3.1.4 and is defined by:

$$\begin{aligned}
 \text{Expressions } \ni e &::= x \mid \text{cst}_\tau \mid \text{null} \mid \text{this} \mid \text{op}(\bar{e}) \mid \text{new } C(\bar{e}) \mid x.m(\bar{e}) \\
 \text{Instructions } \ni I &::= \text{skip}; \mid x:=e; \mid I_1 I_2 \mid \text{while}(e)\{I\} \mid \text{loop}(x)\{I\} \mid \text{if}(e)\{I_1\}\text{else}\{I_2\} \\
 \text{Methods } \ni m_C &::= m(x_1, \dots, x_n)\{I \text{ return } x;\} \\
 \text{Constructors } \ni k_C &::= C(x_1, \dots, x_n)\{X_1:=x_1, \dots, X_n:=x_n\} \\
 \text{Classes } \ni C &::= \text{class } C \{ \overline{\text{var } X}; \overline{k_C} \overline{m_C} \}.
 \end{aligned}$$

The distinctions are that methods always return, that method calls are performed on variables, and that an extra-loop instruction is included in the language. As a consequence, method calls are considered to be expressions. The variable guarding a loop cannot be assigned to within the loop body. In the constructor C , the variable X_1, \dots, X_n have to match exactly the attributes $\overline{\text{var } X}$; in the declaration $C \{ \overline{\text{var } X}; \overline{k_C} \overline{m_C} \}$ of the corresponding class. Moreover, to simplify the discussion, local variables, overload, override, and inheritance and more generally name clashes are prohibited. We also assume that methods are non-recursive.

A main class is a special class called main with no constructor and no method (definitions).

A program is composed by a set of classes with exactly one main class. All the programs have to enjoy some well-formedness conditions that are similar to the well-formedness conditions of Subsection 3.1.4.

Example 3.3.8. Consider the linked list class LL defined below. X and Y represent the head and tail attributes. W and Z are extra-attributes storing intermediate computations. These latter variables are required to compensate for the absence of local variables.

```

class LL {
  var X; var Y; var W; var Z;

  LL(x, y, w, z){X := x; Y := y; W := w; Z := z;}

  getHead(){skip; return X; }

  getTail(){skip; return Y; }

  setTail(y){Y := y; return X; }

  reverse(){
    Z := new LL(X, null, null, nul);
    W := Y;
    loop(Y){
      Z := new LL(W.getHead(), Z, null, null);
      W := W.getTail();
    }
    return Z;
  }
}

```

```

class main {
  var T; var U; var V;
  V := new LL(U, null, null, null);
  loop(T){U := V.setTail(V);}
}
    
```

The semantics however greatly differs as it is a non-reference based semantics. For that purpose, we consider objects as first order values defined by

$$o ::= \text{null} \mid \mathbf{C}(o_1, \dots, o_n).$$

The size of an object is defined as usual. A store σ is a partial map from attributes and parameters to objects. Let \mathbf{I}^n be a shorthand notation for $\mathbf{I} \dots \mathbf{I}$, n times, The big step operational semantics of a program is defined in Figure 3.17, provided that the truth values 1

$$\begin{array}{c}
 \frac{}{(\mathbf{x}, \sigma) \downarrow (\mathbf{x}\sigma, \sigma)} \quad \frac{}{(\text{null}, \sigma) \downarrow (\text{null}, \sigma)} \quad \frac{(\mathbf{e}_i, \sigma) \downarrow (o_i, \sigma)}{(\text{new } \mathbf{C}(\mathbf{e}_1, \dots, \mathbf{e}_n), \sigma) \downarrow (\text{new } \mathbf{C}(o_1, \dots, o_n), \sigma)} \\
 \\
 \frac{\frac{(\mathbf{x}, \sigma) \downarrow (\text{new } \mathbf{C}(o_1, \dots, o_m), \sigma)}{\mathbf{C} \{ \dots \text{var } X_i; \dots \mathbf{m}(\tau_1 \mathbf{x}_1, \dots, \tau_n \mathbf{x}_n) \{ \mathbf{I} \text{ return } \mathbf{y}; \} \dots \}} \quad \frac{\forall j, (\mathbf{e}_j, \sigma) \downarrow (o'_j, \sigma)}{(\mathbf{I}, \sigma \{ X_i \leftarrow o_i, X_j \leftarrow o'_j \}) \downarrow \sigma'} \\
 (\mathbf{x}.\mathbf{m}(\mathbf{e}_1, \dots, \mathbf{e}_n), \sigma) \downarrow (\mathbf{y}\sigma', \sigma' \{ \mathbf{x} \leftarrow \text{new } \mathbf{C}(X_1\sigma', \dots, X_m\sigma') \}) \\
 \\
 \frac{}{(\text{skip};, \sigma) \downarrow \sigma} \quad \frac{(\mathbf{e}, \sigma) \downarrow (o, \sigma')}{(\mathbf{x} := \mathbf{e};, \sigma) \downarrow \sigma' \{ \mathbf{x} \leftarrow o \}} \quad \frac{(\mathbf{e}, \sigma) \downarrow w \in \{0, 1\} \quad (\mathbf{I}_w, \sigma) \downarrow \sigma'}{(\text{if}(\mathbf{e})\{ \mathbf{I}_1 \} \text{else}\{ \mathbf{I}_0 \}, \sigma) \downarrow \sigma'} \\
 \\
 \frac{(\mathbf{I}_1, \sigma) \downarrow \sigma' \quad (\mathbf{I}_2, \sigma') \downarrow \sigma''}{(\mathbf{I}_1 \mathbf{I}_2, \sigma) \downarrow \sigma''} \quad \frac{(\mathbf{e}, \sigma) \downarrow 1 \quad (\mathbf{I} \text{ while}(\mathbf{e})\{ \mathbf{I} \}, \sigma) \downarrow \sigma'}{(\text{while}(\mathbf{e})\{ \mathbf{I} \}, \sigma) \downarrow \sigma'} \\
 \\
 \frac{(\mathbf{e}, \sigma) \downarrow 0}{(\text{while}(\mathbf{e})\{ \mathbf{I} \}, \sigma) \downarrow \sigma} \quad \frac{(\mathbf{x}, \sigma) \downarrow (o, \sigma) \quad (\mathbf{I}^{|\mathbf{o}|}, \sigma) \downarrow \sigma'}{(\text{loop}(\mathbf{x})\{ \mathbf{I} \}, \sigma) \downarrow \sigma'}
 \end{array}$$

Figure 3.17: Big step operational semantics of OO programs

and 0 are encoded by the object `null` and any object distinct from `null`, respectively.

Example 3.3.9 ([MP08a]). Consider the program of Example 3.3.8. For any store σ such that $\mathbf{U}\sigma = o$ and $\mathbf{T}\sigma = o'$, we have:

$$\begin{aligned}
 & (\mathbf{V} := \text{new LL}(\mathbf{U}, \overline{\text{null}});, \sigma) \downarrow \sigma \{ \mathbf{V} \leftarrow \text{new LL}(o, \overline{\text{null}}) \} = \sigma', \\
 & (\mathbf{U} := \mathbf{V}.\text{setTail}(\mathbf{V});, \sigma') \downarrow \sigma \{ \mathbf{V} \leftarrow \text{new LL}(o, \text{new LL}(o, \overline{\text{null}}), \overline{\text{null}}) \}, \\
 & (\text{loop}(\mathbf{T})\{ \mathbf{U} := \mathbf{V}.\text{setTail}(\mathbf{V}); \}, \sigma') \downarrow \sigma \{ \mathbf{V} \leftarrow \mathbf{f}^{|\mathbf{o}'|+1}(\text{null}) \},
 \end{aligned}$$

where $\mathbf{f} = \mathbf{x} \mapsto \text{new LL}(o, \mathbf{x}, \text{null}, \text{null})$ and \mathbf{f}^n is defined in a standard manner.

We can now adapt the notion of assignment in the context of OO programs.

Definition 3.3.4 (Assignment). Given an OO program, an assignment $[-]_{\mathbb{R}^+}$ maps:

- every variable \mathbf{x} to a variable $[\mathbf{x}]_{\mathbb{R}^+}$ in \mathbb{R}^+ ,
- every constructor symbol \mathbf{C} corresponding to a class of n attributes to a total monotonic function $[\mathbf{C}]_{\mathbb{R}^+} : (\mathbb{R}^+)^n \rightarrow \mathbb{R}^+$,
- every method symbol \mathbf{m} with n parameters to a total monotonic function $[\mathbf{m}]_{\mathbb{R}^+} : (\mathbb{R}^+)^{n+1} \rightarrow \mathbb{R}^+$.

Assignments are extended canonically to expressions by:

$$\begin{aligned} [\mathbf{new\ C}(e_1, \dots, e_n)]_{\mathbb{R}^+} &= [\mathbf{C}]_{\mathbb{R}^+}([\mathbf{e}_1]_{\mathbb{R}^+}, \dots, [\mathbf{e}_n]_{\mathbb{R}^+}) \\ [\mathbf{x.m}(e_1, \dots, e_n)]_{\mathbb{R}^+} &= [\mathbf{m}]_{\mathbb{R}^+}([\mathbf{e}_1]_{\mathbb{R}^+}, \dots, [\mathbf{e}_n]_{\mathbb{R}^+}, [\mathbf{x}]_{\mathbb{R}^+}) \end{aligned}$$

The notion of additive and polynomial assignment can be defined as in Subsection 2.2.2.

Definition 3.3.5 (Sup-interpretation). *Given an OO program, an assignment $[-]_{\mathbb{R}^+}$ is a sup-interpretation if for each method \mathbf{m} of arity n and each store σ such that $(\mathbf{x.m}(\mathbf{x}_1, \dots, \mathbf{x}_n), \sigma) \downarrow (o, \sigma')$, we have:*

$$[\mathbf{m}]_{\mathbb{R}^+}([\mathbf{x}_1\sigma]_{\mathbb{R}^+}, \dots, [\mathbf{x}_n\sigma]_{\mathbb{R}^+}, [\mathbf{x}\sigma]_{\mathbb{R}^+}) \geq \max([\mathbf{o}]_{\mathbb{R}^+}, [\mathbf{x}\sigma']_{\mathbb{R}^+}).$$

The above definition is very similar to Definition 2.4.1 of Section 2.4. The main distinction is that, not only the return value sup-interpretation $[\mathbf{o}]_{\mathbb{R}^+}$ is bounded from above, but also the side-effect $[\mathbf{x}\sigma']_{\mathbb{R}^+}$.

The absence of references makes the sup-interpretation to roughly approximate complex data structures such as cyclic data structure. This is not a serious drawback as our goal is to approximate the number of instances, which is preserved by the representation of objects by terms.

Example 3.3.10 ([MP08a]). *Consider the method `setTail` of Example 3.3.8 and define*

$$\begin{aligned} [\mathbf{null}]_{\mathbb{R}^+} &= 0, & [\mathbf{LL}]_{\mathbb{R}^+}(X, Y, W, Z) &= X + Y + W + Z + 1, \\ [\mathbf{setTail}]_{\mathbb{R}^+}(Y, Z) &= Y + Z. \end{aligned}$$

Consider a store σ such that $\mathbf{x}\sigma = \mathbf{new\ LL}(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d})$ and $\mathbf{y}\sigma = 1$. $(\mathbf{x.setTail}(\mathbf{y}), \sigma) \downarrow (\mathbf{a}, \sigma\{\mathbf{x} \leftarrow \mathbf{new\ LL}(\mathbf{a}, 1, \mathbf{c}, \mathbf{d})\})$ holds and we check that:

$$\begin{aligned} [\mathbf{setTail}]_{\mathbb{R}^+}([\mathbf{1}]_{\mathbb{R}^+}, [\mathbf{new\ LL}(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d})]_{\mathbb{R}^+}) &= [\mathbf{1}]_{\mathbb{R}^+} + [\mathbf{new\ LL}(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d})]_{\mathbb{R}^+}, \\ &\geq \max([\mathbf{a}]_{\mathbb{R}^+}, [\mathbf{new\ LL}(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d})]_{\mathbb{R}^+}). \end{aligned}$$

Consequently, it defines a (partial, as we do not provide the full assignment) sup-interpretation.

Now we can develop a criterion on OO programs. For that purpose, we need to introduce a notion of weight.

Definition 3.3.6. *Given a program having a main class with n attributes $\mathbf{X}_1, \dots, \mathbf{X}_n$, a loop command is a loop that does not contain any while loop. A while command is either a while loop or a loop that contains at least one while loop. A weight ω associates to each loop command \mathbf{I} , a total, monotonic, and subterm function $\omega(\mathbf{I})$. A weight is max-polynomial if for every loop command \mathbf{I} , $\omega(\mathbf{I})$ is a max-polynomial.*

Following [MP08a], we now define a criterion called brotherly ensuring that the size of the objects held by the attributes remains polynomially bounded within loops and while loops.

Definition 3.3.7. A program having a main class with n attributes $\mathbf{X}_1, \dots, \mathbf{X}_n$ is brotherly if there are a total, polynomial, and additive sup-interpretation $[-]_{\mathbb{R}^+}$ and a (max-)polynomial weight ω such that

- for each loop command \mathbf{I} of the main class:

– for every method call $a = \mathbf{X}_j.\mathbf{m}(e_1, \dots, e_m)$ in \mathbf{I} :

$$\omega(\mathbf{I})(T + 1, [\mathbf{X}_1]_{\mathbb{R}^+}, \dots, [\mathbf{X}_n]_{\mathbb{R}^+}) \geq \omega(\mathbf{I})(T, [\mathbf{X}_1]_{\mathbb{R}^+}, \dots, [\mathbf{X}_{j-1}]_{\mathbb{R}^+}, [a]_{\mathbb{R}^+}, [\mathbf{X}_{j+1}]_{\mathbb{R}^+}, [\mathbf{X}_n]_{\mathbb{R}^+}),$$

with T a fresh variable.

– for every assignment $\mathbf{X}_j := \mathbf{e}$; in \mathbf{I} :

$$\omega(\mathbf{I})(T + 1, [\mathbf{X}_1]_{\mathbb{R}^+}, \dots, [\mathbf{X}_n]_{\mathbb{R}^+}) \geq \omega(\mathbf{I})(T, [\mathbf{X}_1]_{\mathbb{R}^+}, \dots, [\mathbf{X}_{j-1}]_{\mathbb{R}^+}, [\mathbf{e}]_{\mathbb{R}^+}, [\mathbf{X}_{j+1}]_{\mathbb{R}^+}, [\mathbf{X}_n]_{\mathbb{R}^+}),$$

with T a fresh variable.

- for each while command \mathbf{I} of the main class and each variable assignment $\mathbf{X}_j := \mathbf{e}$; in \mathbf{I} ,

$$\max([\mathbf{X}_1]_{\mathbb{R}^+}, \dots, [\mathbf{X}_n]_{\mathbb{R}^+}) \geq [\mathbf{e}]_{\mathbb{R}^+}.$$

We can now state soundness result on brotherly programs, stating that any object computed by such a program has size polynomially bounded in the input size.

Theorem 3.3.3 ([MP08a]). Consider a brotherly program, whose main class is of the shape `class main {Var \mathbf{X}_1 ; ... Var \mathbf{X}_n ; \mathbf{I} }`, there exists a polynomial P such that for any store σ , if $(\mathbf{I}, \sigma) \downarrow \sigma'$ then

$$P(|\mathbf{X}_1\sigma|, \dots, |\mathbf{X}_n\sigma|) \geq \max_{i=1}^n |\mathbf{X}_i\sigma'|$$

Example 3.3.11 ([MP08a]). Consider the following program that makes use of the class of Example 3.3.8.

```

class main {
  Var X; Var Y; Var Z;
 $\mathbf{I}$ : loop(Z){
      X := Y.reverse();
    }
  Y := X.setTail(Z);
}
    
```

\mathbf{I} is the only loop command and there is no while command. Consequently, for the program to be brotherly, we have to find a polynomial weight ω and a polynomial and additive sup-interpretation $[-]_{\mathbb{R}^+}$ such that:

$$\omega(\mathbf{I})(T + 1, [\mathbf{X}]_{\mathbb{R}^+}, [\mathbf{Y}]_{\mathbb{R}^+}, [\mathbf{Z}]_{\mathbb{R}^+}) \geq \omega(\mathbf{I})(T, [\mathbf{X}]_{\mathbb{R}^+}, [\mathbf{Y.reverse}()]_{\mathbb{R}^+}, [\mathbf{Z}]_{\mathbb{R}^+}),$$

$$\omega(\mathbf{I})(T + 1, [\mathbf{X}]_{\mathbb{R}^+}, [\mathbf{Y}]_{\mathbb{R}^+}, [\mathbf{Z}]_{\mathbb{R}^+}) \geq \omega(\mathbf{I})(T, [\mathbf{Y.reverse}()]_{\mathbb{R}^+}, [\mathbf{Y}]_{\mathbb{R}^+}, [\mathbf{Z}]_{\mathbb{R}^+}).$$

We let the reader check that the assignment $[-]_{\mathbb{R}^+}$ defined by $[\mathbf{reverse}]_{\mathbb{R}^+}(Y) = Y$ and extended with the assignment of Example 3.3.10 defines a total (i.e. defined for every method symbol), polynomial, and additive sup-interpretation. Moreover, setting $\omega(\mathbf{I})(T, X, Y, Z) = Y \times T + X + Y + Z$, we obtain that this program is brotherly as the above inequalities are satisfied. Consequently, by Theorem 3.3.3, all the computed objects are polynomially bounded in the input size.

This subsection has demonstrated that interpretation methods can be gently adapted to OO programs. However the extension has a weak expressive power with respect to the tier based methods of Section 3.1 due to the fact that while loops are treated as non-size increasing computations (see [Hof99, Hof02]) in the definition of the brotherly criterion. This explains partly the reason why most of the advances on OO programs have been obtained using the tier-based techniques of Section 3.1.

3.3.4 Miscellaneous

In this subsection, we discuss other extensions of interpretations to bytecode, concurrent systems, and polygraphs.

Bytecode

The paper [ACGDZJ04] has studied a setting where functional programs in pre-compiled form are exchanged by untrusted parties. It provides code annotations to guarantee type, size, and termination properties at the bytecode level on a simple stack machine. The size upper-bounds are obtained through the use of (polynomial) quasi-interpretations. They basically provide upper-bounds of the stack frame size of the interpreted bytecode.

Concurrent systems

The notion of quasi-interpretation has also been extended in [ADZ04] to statically infer upper bounds on the resources needed for the execution of synchronous cooperative threads. It is combined with termination techniques to guarantee that each instant terminates and to infer a polynomial bound on the resources needed for the execution of the system during an instant in the size of the parameters at the beginning of the instant. This work has been extended in [AD06], obtaining a polynomial bound on the size of the parameters of the program for arbitrarily many instants. It has been extended to ensure feasible reactivity in the framework of the synchronous pi-calculus in [AD06].

Polygraphs

In [BG08, BG07], Bonfante and Guiraud have adapted the interpretation methodology to the context of 3-polygraphs, a family of rewriting systems on circuits with sequential and parallel compositions that can be viewed as a fragment of graph rewriting. They define a notion of *polygraphic program* as a particular subset of 3-polygraphs whose circuit gates (called 2-cells) can be decomposed into function symbols, constructor symbols, and some structural rules and whose rewriting rules (called 3-cells) can be decomposed into computations and structural rules. The structural rules are fully described in [Bon11].

The notion of functorial interpretation is defined on circuits by assigning a non-empty subset of the positive natural numbers to any wire, a weakly monotonic function over the interpretation of wires to each gate of a circuit (sequential composition is interpreted as functional composition and parallel composition as Cartesian product), and by requiring for each rewrite rule that the interpretation of the left-hand side (circuit) is greater or equal to the interpretation of the right-hand side (circuit).

As the inequality is non-strict and also for structural reasons, some further requirements are added, leading to the notion of *Cartesian, conservative and 3-cells-compatible, polygraphic interpretation*. The following alternative characterization of FP is obtained.

Theorem 3.3.4 ([Bon11]). *The set of functions computed by confluent and complete polygraph programs having an additive, polynomial, Cartesian, conservative, and 3-cells-compatible polygraphic interpretation is exactly FP.*

This result is interesting from a theoretical perspective as it shows that interpretations can be adapted fruitfully to computational models based on graph rewriting.

3.4 Extensions of other techniques

In this chapter, we have discussed and studied extensions of the main ICC tools (tiering, soft/light logics, and interpretations) to several programming paradigms (imperative, concurrent, OO, ...). We now briefly discuss extensions of other techniques.

An attempt to extend the matrix based typing analysis of imperative programs à la Jones and Kristiansen to an imperative programming language with higher-order functions was developed in [AKM09]. However the expected results are left as open conjectures.

The framework of automatic amortized resource analysis initiated by [HJ03] for first order functional programs and briefly discussed in Section 1.2.5 has been extended to parallel first order functional programs [HS15], lazy functional languages with coinductive data [VJFH15], C programs [CHS15], and probabilistic programs [NCH18]. Though not directly related, the work of [BHMS04] has also provided an assertion format for automatic certification of heap consumption in a low level language.

Several extensions of the COSTA tool discussed in Subsection 1.2.6 have also been considered, non exhaustively, to non cumulative programming languages with garbage collection [AFRD15, AGGZ13], OO programs [AAG⁺12], real-time Java [KSvG⁺12], and concurrent and distributed programs [ACJ⁺18].

Chapter 4

Extensions to infinite data types

Contents

4.1	Streams and quasi-interpretations	138
4.1.1	A first order lazy stream programming language	139
4.1.2	Parameterized quasi-interpretations	141
4.1.3	Stream upper bounds	141
4.1.4	Bounded input/output properties	143
4.2	Complexity class characterizations	145
4.2.1	Type-2 feasible functionals	146
4.2.2	A stream based characterization of type-2 feasible functionals	147
4.2.3	A stream based characterization of polynomial time over the reals	150
4.2.4	A higher-order characterization of feasible functionals at any type	151
4.3	Coinductive datatypes in the light affine lambda calculus	153
4.3.1	Light affine lambda calculus	154
4.3.2	Algebra and coalgebra in System F	155
4.3.3	Algebra in the light affine lambda calculus	159
4.3.4	Coalgebra in the light affine lambda calculus	160
4.4	Alternative results on streams and real numbers	163
4.4.1	Streams, parsimonious types and non-uniform complexity classes	163
4.4.2	Function algebra characterizations of polynomial time over the reals	167
4.4.3	The BSS model	168

Another line of research in ICC was to try to extend the language with new features and constructs. An extension of interest is the ability to add coinductive data such as infinite lists or infinite trees. The introduction of possibly infinite data raises the difficulty to adapt the corresponding ICC tools that are most of the time based on well-founded properties. Consequently, quasi-interpretations can be viewed a natural candidate to overcome this problem.

Coinductive properties of programming languages and, more generally coinduction, have been deeply studied from a categorical point of view [JR97]. In general, in order to generate and manipulate coinductive data, one typically uses a programming language based on corecursive functions in conjunction with a lazy evaluation strategy. Streams are one kind of coinductive data of interest that allows to represent infinite lists. A main property of streams is the notion of *productivity* [Dij80]. A stream program is productive if it can be evaluated continuously in such a

way that a uniquely determined stream is obtained as the limit. This property is undecidable and several works have been carried out to enforce stream productivity, *e.g.* [Sij89, Coq94, HPS96, EGH⁺10, AM13a, CBGB15, Sev17].

In this section, we are interested in complexity properties of streams and, hence, we will assume to be in a productive setting. The complexity properties we will focus on are quantitative properties such as upper bounds on the number of reduction steps needed to produce the n -th stream element, upper bounds on the size of the output stream elements, or upper bounds on the number of stream inputs needed to produce one stream output element. Section 4.1 will be devoted to prove such properties on first order programs on streams using interpretation methodology as in [GP09a, GP09b, GP15b]. After introducing a first order language on streams in Subsection 4.1.1, we adapt the notion of quasi-interpretation to this setting in Subsection 4.1.2. Criteria to infer upper bounds and bounded input/output properties are studied in Subsection 4.1.3 and Subsection 4.1.4, respectively.

As a stream of integers can be seen as a sequence of integers, which is a basic way to encode real numbers, *e.g.* Cauchy sequences, an extension of ICC tools to streams could also be adapted to study the complexity programs computing over real numbers. Moreover, as a sequence of integers can also be viewed as a function mapping a natural number, the index, to an integer, these extensions could also naturally be considered to study higher-order complexity classes. These two points will be discussed in Section 4.2. After a brief reminder on type 2 complexity in Subsection 4.2.1, Subsection 4.2.2 provides a stream based characterization of the complexity class BFF_2 . Subsection 4.2.3 provides a stream based characterization of $\text{FP}(\mathbb{R})$, the class of functions computable polynomial time over the real numbers. Finally, Subsection 4.2.4 presents an extension of these results which characterizes (discrete) polynomial time at any order.

Section 4.3 will be devoted to ensure polynomial time normalization properties on a lambda calculus augmented with inductive and coinductive datatypes as in [GP15a]. In other words, we show how to extend the expressive power of polynomial time programming languages based on LAL to coinductive data, without breaking soundness. Some reminders on algebra and coalgebra in System F are presented in Subsection 4.3.2. Algebra in LAL are studied in Subsection 4.3.3 and coalgebra in LAL are studied in Subsection 4.3.4.

Finally, in Section 4.4, we discuss other ICC results obtained on stream languages and on real numbers complexity classes. We study the result obtained by Mazza on a parsimonious stream language characterizing the complexity classes P/poly and L/poly using an affine based type system in Subsection 4.4.1. In Subsection 4.4.2, we study a function algebra based characterization of $\text{FP}(\mathbb{R})$. Finally, we discuss in Subsection 4.4.3, a function algebra based characterization of the class $\text{FP}_{\mathbb{R}}$, an alternative notion of polynomial time over the reals based on the Blum, Shub, and Smale (BSS) computational model [BSS88].

4.1 Streams and quasi-interpretations

The papers [GP09a, GP09b, GP15b] consider a simple first order lazy language and study two classes of *space properties* of programs working on streams by means of interpretation methods.

- *Stream Upper Bounds*: these are properties about the size of each stream element produced by a program.
- *Bounded Input/Output Properties*: these are properties about the number of stream elements produced by a program.

These properties analyze two *dimensions* of programs working on streams. The combination of these properties allows one to study a reasonable class of programs and to obtain the information needed in order to improve the memory management process (in terms of bufferization) of programs working on streams.

4.1.1 A first order lazy stream programming language

The language under consideration is a first order lazy variant of the higher-order functional languages described in Subsection 3.3.2:

$$M, N ::= x \mid c \mid M N \mid \mathbf{letRec} \mathbf{f} = \lambda x_1 \dots \lambda x_{ar(\mathbf{f})}. M \mid \mathbf{case} M \mathbf{of} \ c_1 \ x_1 : M_1 \mid \dots \mid c_n \ x_n : M_n.$$

More precisely, the use of lambda abstraction is restricted to function definition of the shape $\mathbf{letRec} \mathbf{f} = \lambda x_1 \dots \lambda x_{ar(\mathbf{f})}. M$, each function variable \mathbf{f} declared through a \mathbf{letRec} being equipped with a fixed arity $ar(\mathbf{f})$. Moreover, there is no partial application, *i.e.* each function application is of the shape $(\mathbf{letRec} \mathbf{f} = M) M_1 \dots M_{ar(\mathbf{f})}$. In what follows, let \mathbf{p} denote a pattern, *i.e.* $\mathbf{p} = c \ x_1 \dots x_{ar(c)}$, for some constructor symbol c and some variables $x_1, \dots, x_{ar(c)}$. The language is also equipped with a type system, that can be found in [GP15b], to ensure that programs do not go wrong. Let \mathbf{Nat} be the type of unary numbers, α be a type variable, and $[\alpha]$ be the type of streams whose elements are of type α . As usual, $\underline{n} : \mathbf{Nat}$ represents the unary number n . Streams correspond to both finite and infinite lists and are encoded using the constructor symbols $\mathbf{nil} : [\alpha]$ and $\mathbf{cons} : \alpha \rightarrow [\alpha] \rightarrow [\alpha]$.

Let also \perp be a special constructor symbol of zero arity representing errors.

We will write $\mathbf{f} \ x_1 \dots x_n = M$ for $\mathbf{letRec} \mathbf{f} = \lambda x_1 \dots \lambda x_n. M$. We define a lazy operational semantics in Figure 4.1 for the language based on the notion of lazy values. A lazy value is just a term whose top most symbol is a constructor symbol

$$\mathbf{lv} := c \ M_1 \dots M_{ar(c)}.$$

As in previous sections, a (strict) value \mathbf{v} is a term that only contains constructor symbols. A term M evaluates to the lazy value \mathbf{lv} whenever $M \Downarrow \mathbf{lv}$. If $M \Downarrow \mathbf{lv}$ using k pattern matching rules over lists (*i.e.* k times a rule (pm) with a conclusion of the shape $\mathbf{case} M \mathbf{of} \ \mathbf{nil} : M_1 \mid \mathbf{cons} \ x \ y : M_2$) then we write $M \Downarrow^k \mathbf{lv}$. A term M evaluates to the value \mathbf{v} , noted $M \Downarrow_v \mathbf{v}$, if the strict evaluation of the term M is equal to the value \mathbf{v} . In other words, $\mathbf{eval} \ M \Downarrow \mathbf{v}$ where \mathbf{eval} is defined for any type α by:

$$\begin{aligned} \mathbf{eval} &: \alpha \rightarrow \alpha, \\ \mathbf{eval} \ (c \ x_1 \dots x_n) &= \hat{C} \ (\mathbf{eval} \ x_1) \dots (\mathbf{eval} \ x_n), \end{aligned}$$

where \hat{C} is a function symbol representing the *strict* version of the primitive constructor c . For instance in the case where c is $+1$ we can define \hat{C} to be the function $\mathbf{succ} :: \mathbf{Nat} \rightarrow \mathbf{Nat}$ defined as:

$$\begin{aligned} \mathbf{succ} \ 0 &= 0+1, \\ \mathbf{succ} \ (x+1) &= (x+1)+1. \end{aligned}$$

We also write $M \Downarrow_v^k \mathbf{v}$ whenever $\mathbf{eval} \ M \Downarrow^k \mathbf{v}$.

Throughout this section, we will use the notation

$$\begin{aligned} \mathbf{f} \ p_1^1 \dots p_n^1 &= M_1 \\ &\vdots \\ \mathbf{f} \ p_1^k \dots p_n^k &= M_k \end{aligned}$$

4.1.2 Parameterized quasi-interpretations

We are now ready to adapt the notion of quasi-interpretation described in Section 2.3 to first order programs on streams. For that purpose, we need to introduce a notion of parameterized quasi-interpretation to be able to deal with local properties of streams such as local upper bounds, properties relating the size of the stream output in a term with respect to its index. Here the parameter is used to take the index into account. Notice that it could be seen as a quasi-interpretation variant and a first order restriction of the higher-order interpretations of Subsection 3.3.2 extended by a parameter.

Definition 4.1.1 (Parameterized assignment). *Given a term M , a parameterized assignment $[-]_{\mathbb{R}^+}^l$ over \mathbb{R}^+ maps:*

- every variable $x \in \mathbb{V}$ to a variable $[x]_{\mathbb{R}^+}^l$ in \mathbb{R}^+ ,
- every symbol $b \in \mathcal{C} \uplus \mathcal{F}$ to a total function $[b]_{\mathbb{R}^+}^1 : \mathbb{R}^{+\text{ar}(b)} \rightarrow \mathbb{R}^+$.

and is extended to the language constructs as follow:

- $[M N]_{\mathbb{R}^+}^l := [M]_{\mathbb{R}^+}^l [N]_{\mathbb{R}^+}^l, M \neq \mathbf{cons}$,
- $[\mathbf{cons} M_1 M_2]_{\mathbb{R}^+}^l := [\mathbf{cons}]_{\mathbb{R}^+}^l [M_1]_{\mathbb{R}^+}^l [M_2]_{\mathbb{R}^+}^{l-1}$,
- $[\mathbf{letRec} f = \lambda x_1 \dots \lambda x_{\text{ar}(f)}. M]_{\mathbb{R}^+} := [f]_{\mathbb{R}^+}^l$,
- $[\mathbf{case} M \text{ of } p_1 : M_1 \mid \dots \mid p_m : M_m]_{\mathbb{R}^+}^l := \max_{1 \leq i \leq m} \{ [M_i]_{\mathbb{R}^+}^l \mid [M]_{\mathbb{R}^+}^l \geq [p_i]_{\mathbb{R}^+}^l \}$.

A parameterized assignment is monotonic if it is monotonic in each argument and in the parameter. The notions of additive and polynomial parameterized assignment are defined in a standard way as in Subsection 2.2.2. We set all the additive constant to 1 throughout the following Subsection. A parameterized assignment is *almost-additive* if it is additive for any constructor symbol distinct from the stream constructor symbol \mathbf{cons} .

Definition 4.1.2 (Parameterized quasi-interpretation). *A term M admits a parameterized quasi-interpretation $[M]_{\mathbb{R}^+}^l$ if the parameterized assignment $[-]_{\mathbb{R}^+}^l$ is monotonic and for each function definition $f \ p_1 \ \dots \ p_n = N$ in M , the following holds:*

$$[f \ p_1 \ \dots \ p_n]_{\mathbb{R}^+}^l \geq [N]_{\mathbb{R}^+}^l.$$

In the particular case where $[-]_{\mathbb{R}^+}^l$ does not depend on the parameter then it is called a standard quasi-interpretation, noted $[-]_{\mathbb{R}^+}$.

A result similar to Lemma 3.3.2, holds.

Lemma 4.1.1 ([GP15b]). *Given a term M admitting the parameterized interpretation $[M]_{\mathbb{R}^+}^l$, if $M \Downarrow \mathbf{1v}$ then $[M]_{\mathbb{R}^+}^l \geq [\mathbf{1v}]_{\mathbb{R}^+}^l$. Moreover if $M \Downarrow^k \mathbf{1v}$ then $[M]_{\mathbb{R}^+}^l \geq [\mathbf{1v}]_{\mathbb{R}^+}^{l-k}$.*

4.1.3 Stream upper bounds

In order to process stream data in a memory-efficient way it is useful to obtain an estimate of the memory needed to store the elements produced by a stream program.

In some situations, an estimate can be obtained by considering in a global way the greatest size of the elements produced by the program as outputs. In other situations, however, there is no such maximal element with respect to the size measure. As a consequence, only an estimate considering the local position of the element in the stream can be given.

Example 4.1.1. Consider the following stream definitions.

$$\begin{array}{ll} \mathbf{ones} : [\mathbf{Nat}] & \mathbf{nats} : \mathbf{Nat} \rightarrow [\mathbf{Nat}] \\ \mathbf{ones} = \mathbf{cons} \ \underline{1} \ \mathbf{ones} & \mathbf{nats} \ x = \mathbf{cons} \ x \ (\mathbf{nats} \ (x+1)) \end{array}$$

Indeed, in the stream definition of **ones** all the elements have the same size, while in the definition of **nats** every element has a size depending on its position in the stream. **ones** has a global upper bound whereas for any \underline{n} , **nats** \underline{n} has a local upper bound.

We provide a formal definition of those properties in the following definition.

Definition 4.1.3 (Local and Global Upper Bounds). A stream program $\mathbf{M} : [\alpha]$ has a local upper bound if there is a function $F \in \mathbb{R}^+ \rightarrow \mathbb{R}^+$ such that

$$\forall \underline{n}, \text{ if } !! \mathbf{M} \ \underline{n} \Downarrow_v \ \mathbf{v} \text{ then } F(|\underline{n}|) \geq |\mathbf{v}|$$

If the function F is constant, then $\mathbf{M} : [\alpha]$ has also a global upper bound.

Definition 4.1.4 (Linear program). Let \Downarrow^k and \Downarrow_v^k be the relations defined by $\mathbf{M} \Downarrow^k \ \mathbf{1v}$ if there exists $k' \leq k$ such that $\mathbf{M} \Downarrow^{k'} \ \mathbf{1v}$ and $\mathbf{M} \Downarrow_v^k \ \mathbf{v}$ if there exists $k' \leq k$ such that $\mathbf{M} \Downarrow_v^{k'} \ \mathbf{v}$, respectively. A program $\mathbf{M} : [\alpha]$ is linear if there is a $k \geq 1$ such that for all $\underline{n} \in \mathbf{Nat}$, $!! \mathbf{M} \ \underline{n} \Downarrow_v^{k \times (|\underline{n}|+1)} \ \mathbf{v}$ holds.

Now we are ready to define two criteria for ensuring the two upper bounds.

Definition 4.1.5 (LUB). A program $\mathbf{M} : [\alpha]$ is LUB if it is linear and it admits a parameterized quasi-interpretation $[-]_{\mathbb{R}^+}^l$ that is almost-additive and such that

$$[\mathbf{cons}]_{\mathbb{R}^+}^l(X, Y) = \max(X, Y).$$

Definition 4.1.6 (GUB). A program $\mathbf{M} : [\alpha]$ is GUB if it admits a standard quasi-interpretation $[-]_{\mathbb{R}^+}$ that is almost-additive and such that

$$[\mathbf{cons}]_{\mathbb{R}^+}(X, Y) = \max(X, Y).$$

Theorem 4.1.1 ([GP15b]). If a program is LUB (GUB) then it admits a local (global) upper bound.

Example 4.1.2. Consider the stream definition of **nats** of Example 4.1.1. We want to show that **nats** is LUB. First, notice that the program **nats** is linear with linearity constant $k = 1$. Indeed, the definition of **nats** does not involve pattern matching on stream data so the only possible pattern matching corresponds to the $!!$ definition where one read is needed to produce one output. Now, consider the parameterized quasi-interpretation $[-]_{\mathbb{R}^+}^l$ defined by: $[\mathbf{nats}]_{\mathbb{R}^+}^l(X) = X + l$, $[+1]_{\mathbb{R}^+}^l(X) = X + 1$, $[0]_{\mathbb{R}^+}^l = 0$, and $[\mathbf{cons}]_{\mathbb{R}^+}^l(X, Y) = \max(X, Y)$. We check that $\forall l \in \mathbb{R}$:

$$\begin{aligned} [\mathbf{nats} \ x]_{\mathbb{R}^+}^l &= [\mathbf{nats}]_{\mathbb{R}^+}^l \ [x]_{\mathbb{R}^+}^l = [x]_{\mathbb{R}^+}^l + l \\ &\geq \max([x]_{\mathbb{R}^+}^l, ([x]_{\mathbb{R}^+}^l + 1) + (l - 1)) \\ &\geq \max([x]_{\mathbb{R}^+}^l, [\mathbf{nats}(x+1)]_{\mathbb{R}^+}^{l-1}) \\ &\geq [\mathbf{cons} \ x \ (\mathbf{nats} \ (x+1))]_{\mathbb{R}^+}^l. \end{aligned}$$

The quasi-interpretation $[-]_{\mathbb{R}^+}^l$ clearly respects the required criterion for **nats** to be LUB: it is monotonic and almost-additive as $[\mathbf{cons}]_{\mathbb{R}^+}^l(X, Y) = \max(X, Y)$.

So, **nats** admits a local upper bound. We obtain the required bound by setting $F(X) = [\mathbf{nats}(\underline{m})]_{\mathbb{R}^+}^X = X + [\underline{m}]_{\mathbb{R}^+}^X = X + |\underline{m}|$. For each unary numbers $\underline{m}, \underline{n} \in \mathbf{Nat}$ such that $\mathbf{eval}((\mathbf{nats} \ \underline{m}) \ !! \ \underline{n}) \Downarrow_v \ \mathbf{v}_n$, the following holds $F(|\underline{n}|) \geq |\underline{n}| + |\underline{m}| \geq |\mathbf{v}_n|$. Indeed for all $\underline{n}, \mathbf{v}_n = \underline{m} + \underline{n}$ and $|\underline{m} + \underline{n}| = |\underline{n}| + |\underline{m}|$.

Example 4.1.3. Consider expressions built using the following function definitions.

```

repeat : Nat → [Nat]
repeat x = cons x (repeat x)

zip : [α] → [α] → [α]
zip (cons x xs) ys = cons x (zip ys xs)

square : [Nat] → [Nat]
square (cons x xs) = cons (mul x x) (square xs)

mul : Nat → Nat → Nat
mul (x+1) y = add y (mul x y)
mul 0 y = 0

```

Each program will only produce output stream elements whose size is bounded by some constant k . For instance, the program

$$\text{square (zip (repeat } \underline{5}) \text{ (square (zip (repeat } \underline{7}) \text{ (repeat } \underline{4}))))}$$

computes stream elements whose size is bounded by $k = 2401 = 7^4$. So, its global upper bound is given by the constant $k = 2401$. Now, consider the following generalization of the above program

$$\text{square (zip (repeat } \underline{5}) \text{ (square (zip } x \text{ (repeat } \underline{4}))))}.$$

It is easy to verify that when x is substituted by a stream s having element sizes bounded by a constant k , then the output stream will have only elements whose sizes are bounded either by 4^4 or by k^4 . So this expression has a global upper bound given by the function F defined by $F(X) = \max(256, X^4)$.

We can show that it is GUB with respect to the quasi-interpretation $[-]_{\mathbb{R}^+}$ defined by:

$$\begin{aligned}
[0]_{\mathbb{R}^+} &= 0, \\
[+1]_{\mathbb{R}^+}(X) &= X + 1, \\
[\text{add}]_{\mathbb{R}^+}(X, Y) &= X + Y, \\
[\text{zip}]_{\mathbb{R}^+}(X, Y) &= [\text{cons}]_{\mathbb{R}^+}(X, Y) = \max(X, Y), \\
[\text{square}]_{\mathbb{R}^+}(X) &= X^2, \\
[\text{mul}]_{\mathbb{R}^+}(X, Y) &= X \times Y, \\
[\text{repeat}]_{\mathbb{R}^+}(X) &= X.
\end{aligned}$$

Indeed, we can check the inequalities of the quasi-interpretation are satisfied for every definition and for every variable assignment.

4.1.4 Bounded input/output properties

Another information of interest in order to improve memory-efficiency is an estimate of the number of elements produced by a stream program when fed with only a restricted portion of the input stream.

In some situations, such an estimate can be obtained by considering only the *length* of the portion of the input stream. In other situations, however, this is not sufficient and so in order to obtain the estimate one needs to consider also the *size* of the elements in the portion.

Example 4.1.4. Consider the following definitions

$$\begin{aligned} \text{merge} & : [\alpha] \rightarrow [\alpha] \rightarrow [a \times a] \\ \text{merge} (\text{cons } x \text{ } xs) (\text{cons } y \text{ } ys) & = \text{cons } (x, y) (\text{merge } ys \text{ } xs) \\ \text{dup} & : [\alpha] \rightarrow [\alpha] \\ \text{dup} (\text{cons } x \text{ } xs) & = \text{cons } x (\text{cons } x (\text{dup } xs)) \end{aligned}$$

Each stream expression built using only `merge` and `dup` will only generate a number of output elements that depends on the number of input read elements. For example, the expression `dup (merge (dup s) (dup s))` produces for each read element of the input stream `s` four elements of the type $a \times a$. In what follows, we will study a Length Based I/O Upper Bound criterion ensuring that the number m of output stream elements is bounded by a function in the number n of stream elements read on the input.

There are stream functions that generate a number of output elements also depends on the size of the input read elements.

Example 4.1.5. Consider the following definitions:

$$\begin{aligned} \text{app} & : [\alpha] \rightarrow [\alpha] \rightarrow [\alpha] & \text{upto} & : \text{Nat} \rightarrow [\text{Nat}] \\ \text{app} (\text{cons } x \text{ } xs) \text{ } ys & = \text{cons } x (\text{app } xs \text{ } ys) & \text{upto } 0 & = \text{nil} \\ \text{app } \text{nil } \text{ } ys & = ys & \text{upto } (x+1) & = \text{cons } (x+1) (\text{upto } x) \\ \text{extendupto} & : [\text{Nat}] \rightarrow [\text{Nat}] \\ \text{extendupto} (\text{cons } x \text{ } xs) & = \text{app } (\text{upto } x) (\text{extendupto } xs) \end{aligned}$$

Every stream expression built using only `upto` and `extendupto` will generate a number of output elements that is related to both the number and the size of input read elements. For example, the expression `extendupto (extendupto s)` outputs $\sum_{i=1}^{|\underline{n}|} i$ elements, for each natural number \underline{n} in the input stream `s`. In what follows, we will also study a Size-Based I/O Upper Bound criterion ensuring that the number m of output stream elements is bounded by a function in the number and the size of the stream elements in input.

A stream function is a term `M` from stream to stream with one free variable. By abuse of notation, we denote such a term by $\lambda x.M$ to name explicitly the free variable `x` in `M`. Indeed, the use of lambda abstractions is restricted in our first order language.

Definition 4.1.7 (Length and size based I/O upper bounds).

- A stream function $\lambda x.M : [\alpha] \rightarrow [\beta]$ has a length based I/O upper bound if there is a function $F : \mathbb{R}^+ \rightarrow \mathbb{R}^+$ such that for every expression `s` : $[\alpha]$:

$$\forall \underline{n} \in \text{Nat}, \text{ if } \text{take } \underline{n} \text{ } s \Downarrow_v v \text{ implies } \text{lg } M\{v/x\} \Downarrow_v \underline{m} \text{ then } F(|\underline{n}|) \geq |\underline{m}|.$$

- A stream function $\lambda x.M : [\alpha] \rightarrow [\beta]$ has a size based I/O upper bound if there is a function $F : \mathbb{R}^+ \rightarrow \mathbb{R}^+$ such that for every expression `s` : $[\alpha]$:

$$\forall \underline{n} \in \text{Nat}, \text{ if } \text{take } \underline{n} \text{ } s \Downarrow_v v \text{ implies } \text{lg } M\{v/x\} \Downarrow_v \underline{m} \text{ then } F(|v|) \geq |\underline{m}|.$$

We are now ready to define two restrictions on quasi-interpretations to ensure the above properties.

Definition 4.1.8 (LBUB and SBUB). A stream function is LBUB (respectively SBUB) if it admits an almost-additive (respectively additive) standard quasi-interpretation $[-]_{\mathbb{R}^+}$ such that $[+1]_{\mathbb{R}^+}(X) = X + 1$ and $[\text{cons}]_{\mathbb{R}^+}(X, Y) = Y + 1$ (respectively $[\text{cons}]_{\mathbb{R}^+}(X, Y) = X + Y + 1$).

Theorem 4.1.2 ([GP15b]). If a stream function $\lambda x.M$ is LBUB (SBUB, respectively) then it has a length (respectively size) based I/O upper bound.

Example 4.1.6. Consider again the stream definitions presented in Example 4.1.4:

```
merge : [α] → [α] → [α]
merge (cons x xs) (cons y ys) = cons (x, y) (merge ys xs)

dup : [α] → [α]
dup (cons x xs) = cons x (cons x (dup xs))
```

To verify that every expression built using these definitions has a length based I/O upper bound it is sufficient to verify that it is LBUB with respect to the following standard quasi-interpretation:

$$[\text{merge}]_{\mathbb{R}^+}(X, Y) = \max(X, Y), \quad [\text{dup}]_{\mathbb{R}^+}(X) = 2X, \quad \text{and} \quad [\text{cons}]_{\mathbb{R}^+}(X, Y) = Y + 1.$$

As an example, for each s we have:

$$[\text{dup} (\text{merge} (\text{dup } s) (\text{dup } s))]_{\mathbb{R}^+} = 4[s]_{\mathbb{R}^+}$$

and this gives a length based I/O upper bound.

Example 4.1.7. Consider again the stream definitions of Example 4.1.5:

```
app : [α] → [α] → [α]
app (cons x xs) ys = cons x (app xs ys)
app nil ys = ys

upto : Nat → [Nat]
upto 0 = nil
upto (x+1) = cons (x+1) (upto x)

extendupto : [Nat] → [Nat]
extendupto (cons x xs) = app (upto x) (extendupto xs)
```

To show that every expression built using these definitions has a size based I/O upper bound it is enough to show that the program is SBUB with respect to the following standard quasi-interpretation:

$$[\text{nil}]_{\mathbb{R}^+} = [0]_{\mathbb{R}^+} = 0, \quad [+1]_{\mathbb{R}^+}(X) = X + 1, \quad [\text{cons}]_{\mathbb{R}^+}(X, Y) = X + Y + 1,$$

$$[\text{app}]_{\mathbb{R}^+}(X, Y) = X + Y, \quad [\text{upto}]_{\mathbb{R}^+}(X) = [\text{extendupto}]_{\mathbb{R}^+}(X) = 2 \times X^2.$$

In particular, by taking $F(X) = [\text{extendupto}]_{\mathbb{R}^+}([\text{extendupto}]_{\mathbb{R}^+}(X)) = 8 \times X^4$ we obtain a size based I/O upper bound for the expression $\text{extendupto} (\text{extendupto } s)$. The function F also gives a bound also on the size of produced elements.

4.2 Complexity class characterizations

In [FHHP10, FHHP15], second order interpretations were combined to a first order language on streams to characterize the class of type-2 Feasible Functionals, also known as *Basic Feasible*

Functionals BFF_2 , that is the type-2 counterpart of the class of polynomial time computable functions FP .²⁷ In this Section, after recalling some basic background on type-2 feasibility in Subsection 4.2.1, we present the characterization of [FHHP10, FHHP15] in Subsection and extensions characterizing the polynomial time over the reals and at higher-orders in Subsection 4.2.3 and Subsection 4.2.4, respectively.

4.2.1 Type-2 feasible functionals

The notion of feasibility for type-2 functionals was first studied by Constable [Con73] and Mehlhorn [Meh76] using an explicit bound on the size of computations. Later, Cook and Kapron [CK90], Cook [Coo92], Cook and Urquhart [CU93] have provided alternative (non ICC) characterizations of this notion of type-2 polynomial time complexity based on programming languages with explicit bounds, machines, and recursion scheme, respectively.

Other (non ICC) characterizations of BFF_2 have been provided in the literature. The characterizations of [IRK01] are based on a simple imperative programming language that enforces an explicit external bound on the size of oracle outputs within loops. Function algebra characterizations were developed in [KS19]. Several characterizations using type-2 polynomials [KC91, KC96] and type-1 polynomials with *bounded lookahead revisions* [KS18] were also developed using Oracle Turing Machines (OTMs).

In this section, we recall Kapron and Cook characterization of basic polynomial time functionals (BFF) [KC96] based on the notion of OTM.

Definition 4.2.1 (Oracle Turing Machine). *An Oracle Turing Machine \mathcal{M} with k oracles (where oracles are functions from \mathbb{N} to \mathbb{N}) and l input tapes is a TM with, for each oracle, a state, one query tape and one answer tape.*

If \mathcal{M} is used with oracles $F_1, \dots, F_k : \mathbb{N} \rightarrow \mathbb{N}$, then on the oracle state $i \in \{1, \dots, k\}$, $F_i(x)$ is written on the corresponding answer tape, whenever x is the content of the corresponding query tape.

Definition 4.2.2 (Size of a function). *The size $|F| : \mathbb{N} \rightarrow \mathbb{N}$ of a function $F : \mathbb{N} \rightarrow \mathbb{N}$ is defined by:*

$$|F|(n) = \max_{k \leq n} |F(k)|$$

where $|F(k)|$ represents the size of the binary representation of $F(k)$.

Definition 4.2.3 (Second order polynomial). *A second order polynomial is a polynomial generated by the following grammar:*

$$P := c \mid X \mid P + P \mid P \times P \mid Y(P),$$

where X represents a first order variable, Y a second order variable, and c a constant in \mathbb{N} .

The following example shows the implicit meaning of a substitution of a second order variable.

Example 4.2.1. *Given $P(Y, X) = Y(X^2 + 1)$ and $f : \mathbb{N} \rightarrow \mathbb{N}$ define by $f(X) = X^2$, we have $P(f, X) = f(X^2 + 1) = (X^2 + 1)^2$.*

In the following, $P(Y_1, \dots, Y_k, X_1, \dots, X_l)$ will denote a second order polynomial where each Y_i , $1 \leq i \leq k$, represents a second order variable, and each X_j , $1 \leq j \leq l$, a first order variable.

Definition 4.2.4 (Polynomial running time). *The cost of a transition is:*

²⁷This class is also called BFF for short when the type-2 setting is clear from the context.

- $|F|(|x|)$, if the machine is in a query state of the oracle F on input query x ,
- 1 otherwise.

An OTM \mathcal{M} operates in time $T : (\mathbb{N} \rightarrow \mathbb{N})^k \rightarrow \mathbb{N}^l \rightarrow \mathbb{N}$ if for all inputs $x_1, \dots, x_l : \mathbb{N}$ and $F_1, \dots, F_k : \mathbb{N} \rightarrow \mathbb{N}$, the sum of the transition costs before \mathcal{M} halts on these inputs is less than $T(|F_1|, \dots, |F_k|, |x_1|, \dots, |x_l|)$.

A function $G : (\mathbb{N} \rightarrow \mathbb{N})^k \rightarrow \mathbb{N}^l \rightarrow \mathbb{N}$ is OTM computable in polynomial time if there exists a second order polynomial P such that $G(F_1, \dots, F_k, x_1, \dots, x_l)$ is computed by an OTM in time $P(|F_1|, \dots, |F_k|, |x_1|, \dots, |x_l|)$ on inputs x_1, \dots, x_l and oracles F_1, \dots, F_k .

Theorem 4.2.1 ([KC96]). *The set of polynomial time OTM computable functions is exactly the class of type-2 (Basic) Feasible Functionals, BFF_2 .*

4.2.2 A stream based characterization of type-2 feasible functionals

The syntax of the first order language on stream studied in [FHHP10, FHHP15] is very similar to the language of Section 4.1.1.

The main distinction is that the semantics is only lazy on stream data (and strict on other datatypes) that are encoded as in Section 4.1.1 using the constructor symbols `nil` and `cons`. This amounts to considering a reduction \rightarrow that corresponds to \Downarrow on streams of type $[\alpha]$ and to \Downarrow_v on other types (see Section 4.1.1). Consequently, in this context, lazy values are of the shape:

$$\text{lv} := \text{cons } M_1 \ M_2,$$

for some terms M_1 and M_2 .

Using an idea similar to the encoding of a function over natural numbers as a stream of integers, the interpretation of a stream will be a type-1 function from natural numbers to natural numbers. Here we avoid the use of parameterized interpretations to simplify the discussion.

In the following, let positive functionals denote functions in $((\mathbb{N} \rightarrow \mathbb{N})^k \times \mathbb{N}^l) \rightarrow T$ with $k, l \in \mathbb{N}$ and $T \in \{\mathbb{N}, \mathbb{N} \rightarrow \mathbb{N}\}$. Given a positive functional $F : ((\mathbb{N} \rightarrow \mathbb{N})^k \times \mathbb{N}^l) \rightarrow T$, the arity of F is $k + l$.

Let $>$ denote the usual ordering on \mathbb{N} and $\mathbb{N} \rightarrow \mathbb{N}$, i.e. given $F, G : \mathbb{N} \rightarrow \mathbb{N}$, $F > G$ if $\forall X \in \mathbb{N} \setminus \{0\}, F(X) > G(X)$. We extend this ordering to positive functionals of arity l by: $F > G$ if $\forall X_1, \dots, \forall X_l \in \{\mathbb{N} \setminus \{0\}, \mathbb{N} \rightarrow^\uparrow \mathbb{N}\}, F(X_1, \dots, X_l) > G(X_1, \dots, X_l)$, where $\mathbb{N} \rightarrow^\uparrow \mathbb{N}$ is the set of increasing functions on positive integers. A positive functional F of arity n is monotonic if $\forall i \in \{1, n\}, \forall X_i > X'_i, F(\dots, X_i, \dots) > F(\dots, X'_i, \dots)$, where $X_i, X'_i \in \{\mathbb{N} \setminus \{0\}, \mathbb{N} \rightarrow^\uparrow \mathbb{N}\}$.

Definition 4.2.5. *An assignment of a program is a total mapping of the function and constructor symbols to monotonic positive functionals. The type of the interpretation is inductively defined by the type of the corresponding symbol:*

- a symbol \mathbf{b} of type A ($A \neq [\alpha]$) has interpretation $[\mathbf{b}]_{\mathbb{N}}$ in \mathbb{N} ,
- a symbol \mathbf{b} of type $[\alpha]$ has interpretation $[\mathbf{b}]_{\mathbb{N}}$ in $\mathbb{N} \rightarrow \mathbb{N}$,
- a symbol \mathbf{b} of type $A \rightarrow B$ has interpretation $[\mathbf{b}]_{\mathbb{N}}$ in $T_A \rightarrow T_B$, whether T_A and T_B are the types of the interpretations of the symbols of type \mathbf{A} and, respectively, type \mathbf{B} .

The assignments of the constructs `case M of $c_1 \ x_1 : N_1 \dots c_n \ x_n : N_n$ and letRec $f = M$ are defined as in Subsection 4.1.1.`

We fix the assignment of each constructor symbol by:

- $[c]_{\mathbb{N}}(X_1, \dots, X_{ar(c)}) = X_1 + \dots + X_{ar(c)} + 1$, if $c \in \mathcal{C} \setminus \{\mathbf{cons}\}$,
- $[\mathbf{cons}]_{\mathbb{N}}(X, Y)(Z + 1) = 1 + X + Y(Z)$,
- $[\mathbf{cons}]_{\mathbb{N}}(X, Y)(0) = X$.

Once the assignment of each symbol is fixed, we can extend assignments to any expression by structural induction:

- $[x]_{\mathbb{N}} = X$, if x is a variable of type A ($A \neq [\alpha]$) and X is a variable in \mathbb{N} ,
- $[y]_{\mathbb{N}}(Z) = Y(Z)$, if y is a variable of type $[\alpha]$, i.e. we associate a unique second order variable $Y : \mathbb{N} \rightarrow \mathbb{N}$ to each $y \in \mathcal{X}$ of type $[\alpha]$,
- $[t \ e_1 \ \dots \ e_n]_{\mathbb{N}} = [t]_{\mathbb{N}}([e_1]_{\mathbb{N}}, \dots, [e_n]_{\mathbb{N}})$, if t is a constructor or function symbol.

An assignment is polynomial if all symbols are interpreted as functions bounded by type-2 polynomials.

Consequently, an assignment $[-]_{\mathbb{N}}$ maps any expression to a functional (of the assignment of its free variables).

Example 4.2.2. The stream constructor \mathbf{cons} has type $\alpha \rightarrow [\alpha] \rightarrow [\alpha]$. Consequently, its assignment $[\mathbf{cons}]_{\mathbb{N}}$ has type $(\mathbb{N} \times (\mathbb{N} \rightarrow \mathbb{N})) \rightarrow (\mathbb{N} \rightarrow \mathbb{N})$.²⁸ Considering the expression $\mathbf{cons} \ p \ (\mathbf{cons} \ q \ r)$, with p, q, r variables, we obtain that:

$$\begin{aligned} [\mathbf{cons} \ p \ (\mathbf{cons} \ q \ r)]_{\mathbb{N}} &= [\mathbf{cons}]_{\mathbb{N}}([p]_{\mathbb{N}}, [\mathbf{cons} \ q \ r]_{\mathbb{N}}) \\ &= [\mathbf{cons}]_{\mathbb{N}}([p]_{\mathbb{N}}, [\mathbf{cons}]_{\mathbb{N}}([q]_{\mathbb{N}}, [r]_{\mathbb{N}})) \\ &= [\mathbf{cons}]_{\mathbb{N}}(P, [\mathbf{cons}]_{\mathbb{N}}(Q, R)) \\ &= F(P, Q, R) \end{aligned}$$

where $F \in ((\mathbb{N} \rightarrow \mathbb{N}) \times \mathbb{N}^2) \rightarrow (\mathbb{N} \rightarrow \mathbb{N})$ is the positive functional such that:

- $F(P, Q, R)(Z + 2) = 1 + P + [\mathbf{cons}]_{\mathbb{N}}(Q, R)(Z + 1) = 2 + P + Q + R(Z)$,
- $F(P, Q, R)(1) = 1 + P + [\mathbf{cons}]_{\mathbb{N}}(Q, R)(0) = 1 + P + Q$,
- $F(P, Q, R)(0) = P$.

Definition 4.2.6 (Interpretation). The assignment of a term M is an interpretation if for each definition $f \ p_1 \ \dots \ p_n = e \in M$,

$$[f \ p_1 \ \dots \ p_n]_{\mathbb{N}} > [e]_{\mathbb{N}}$$

By extension, a term M admits a (polynomial) interpretation if there exists a (polynomial) assignment that is an interpretation of M .

The following programs are examples of well-founded polynomial programs.

²⁸We will use the Cartesian product instead of the arrow for the argument types of a function symbol in the following.

Example 4.2.3. $!! s \underline{n}$ computes the $(n + 1)^{th}$ element of the stream s

$$\begin{aligned} !! : [\alpha] &\rightarrow \text{Nat} \rightarrow \alpha \\ !! (\text{cons } x \text{ } xs) \ 0 &= x \\ !! (\text{cons } x \text{ } xs) \ (y+1) &= !! \text{ } xs \ y \end{aligned}$$

and admits an interpretation $[!!]_{\mathbb{N}}$ in $((\mathbb{N} \rightarrow \mathbb{N}) \times \mathbb{N}) \rightarrow \mathbb{N}$ defined by $[!!]_{\mathbb{N}}(Y, N) = Y(N)$. Indeed, we check that:

$$\begin{aligned} [!! (\text{cons } x \text{ } xs) \ (y+1)]_{\mathbb{N}} &= [\text{cons } x \text{ } xs]_{\mathbb{N}}([y]_{\mathbb{N}} + 1) = 1 + [x]_{\mathbb{N}} + [xs]_{\mathbb{N}}([y]_{\mathbb{N}}), \\ &> [xs]_{\mathbb{N}}([y]_{\mathbb{N}}) = [!! \text{ } xs \ y]_{\mathbb{N}}, \end{aligned}$$

and

$$[!! (\text{cons } x \text{ } xs) \ 0]_{\mathbb{N}} = [(\text{cons } x \text{ } xs)]_{\mathbb{N}}([0]_{\mathbb{N}}) = [(\text{cons } x \text{ } xs)]_{\mathbb{N}}(1) = 1 + [x]_{\mathbb{N}} + [xs]_{\mathbb{N}}(0) > [x]_{\mathbb{N}}.$$

In the same way, we let the reader check that tln , which drops the first $n + 1$ elements of a stream, admits the well-founded interpretation $[\text{tln}]_{\mathbb{N}}$ of type $((\mathbb{N} \rightarrow \mathbb{N}) \times \mathbb{N}) \rightarrow (\mathbb{N} \rightarrow \mathbb{N})$ defined by $[\text{tln}]_{\mathbb{N}}(Y, N)(Z) = Y(N + Z + 1)$.

$$\begin{aligned} \text{tln} : [\alpha] &\rightarrow \text{Nat} \rightarrow [\alpha] \\ \text{tln} (\text{cons } x \text{ } xs) \ 0 &= xs \\ \text{tln} (\text{cons } x \text{ } xs) \ (y+1) &= \text{tln } xs \ y \end{aligned}$$

Indeed, for the last rule, we just check that $[\text{tln} (\text{cons } x \text{ } xs) \ (y+1)]_{\mathbb{N}} > [\text{tln } xs \ y]_{\mathbb{N}}$, that is $\forall Z \in \mathbb{N} \setminus \{0\}$, $[\text{tln} (\text{cons } x \text{ } xs) \ (y+1)]_{\mathbb{N}}(Z) > [\text{tln } xs \ y]_{\mathbb{N}}(Z)$.

Before stating the main characterization of type-2 Feasible Functionals, we need to restrict the considered interpretations.

Definition 4.2.7 (exp-poly). We call exp-poly the set of functions generated by the following grammar:

$$EP := P \mid EP + EP \mid EP \times EP \mid Y(2^{EP}),$$

where P denotes a first order polynomial and Y a second order variable.

The interpretation of a program is exp-poly if each symbol is interpreted by an exp-poly function.

Theorem 4.2.2 ([FHHP15]). BFF_2 is exactly the set of functions that can be computed by programs that admit a exp-poly interpretation.

Notice that the exponential is due to the fact that the time bound is encoded in [FHHP15] uses unary numbers rather than binary numbers. Indeed a decrease by one on a binary number does not imply a strict decrease of the interpretation. Consequently, this exponential is restricted to values (type-0 data).

4.2.3 A stream based characterization of polynomial time over the reals

Now we show that the above characterization can be adapted to polynomial time over the real numbers, as any real number can be encoded as a stream of integers.

Indeed, type 2 functionals can be used to represent real functions. Recursive analysis models computations on real numbers as computations on converging sequences of rational numbers. For interested readers, [Wei00] provides an in-depth view of recursive analysis from a computability point of view. Complexity in this model is treated in [Ko91].

We will require a given convergence speed to be able to compute efficiently. A real number $x \in \mathbb{R}^+$ is represented by the sequence $(q_n) \in \mathbb{Q}^{\mathbb{N}}$ if $\forall i \in \mathbb{N}, |x - q_i| < 2^{-i}$. This will be denoted by $(q_n) \rightsquigarrow x$. A function $f : \mathbb{R} \rightarrow \mathbb{R}$ will be said to be computed by a machine M if

$$(q_n) \rightsquigarrow x \Rightarrow (M(q_n)) \rightsquigarrow f(x). \quad (4.1)$$

Hence if α is an inductive type describing rational numbers (*e.g.* pairs of binary integers) then a computable real function could be represented by stream programs of type $[\alpha] \rightarrow [\alpha]$, where α is an inductive type describing the set of rationals \mathbb{Q} . Only programs encoding machines verifying implication (4.1) will make sense in this framework. Property (4.1) is highly semantic and hence difficult to enforce using static analysis techniques.

The above definition implies that if a real function is computable then it must be continuous (see [Wei00]).

Definition 4.2.8 (Complexity of a real function). *$f : \mathbb{N} \rightarrow \mathbb{N}$ is a complexity measure for Machine M if and only if $\forall (q_n) \in \mathbb{Q}^{\mathbb{N}}, M(q_n)$ is output in time $f(n)$.*

The continuity property described above has a counterpart in complexity related to the modulus of continuity.

Definition 4.2.9 (Modulus of continuity). *Given $f : \mathbb{R} \rightarrow \mathbb{R}$, we say that $m : \mathbb{N} \rightarrow \mathbb{N}$ is a modulus of continuity for f if and only if*

$$\forall n \in \mathbb{N}, \forall r, s \in \mathbb{R}, |r - s| < 2^{-m(n)} \implies |f(r) - f(s)| < 2^{-n}.$$

Definition 4.2.10. *The class of functions computed by Machines that have polynomial time complexity is denoted $\text{FP}(\mathbb{R})$.*

Proposition 4.2.1. *If $f \in \text{FP}(\mathbb{R})$ then it has a polynomial modulus of continuity.*

This proposition provides an interesting result for functions in $\text{FP}(\mathbb{R})$. Indeed, it tells us that there are polynomial m and P such that for any real number r and any precision n , it suffices to read the first $m(n)$ digits of the stream representation of r and then to perform $P(n)$ computational steps in order to approximate $f(r)$ with a precision 2^{-n} .

Consequently, a stream based programming language allowed to compute a polynomial number of steps on a stream encoding would allow us to simulate such a behaviour.

Remark that the notion of Size based I/O upper bound of Subsection 4.1.4 can be viewed as sufficient condition to force the program to compute a function with a modulus of continuity. Indeed the size based I/O upper bound was defined as

$$\forall \underline{n} \in \text{Nat}, \text{ if } \mathbf{take} \ \underline{n} \ \mathbf{s} \Downarrow_v \ v \text{ implies } \mathbf{lg} \ \mathbf{M}\{\mathbf{v}/\mathbf{x}\} \Downarrow_v \ \underline{m} \text{ then } F(|\underline{n}|) \geq |\underline{m}|.$$

If the stream \mathbf{s} represents a real number (*e.g.* using a representation with digits in $\{-1, 0, 1\}$) then $\mathbf{take} \ \underline{n} \ \mathbf{s}$ provides the \underline{n} first elements of \mathbf{s} and could be an approximation of the fractional

part of the real number (the error being bounded by 2^{-n}). The length of the output is bounded by $F(\mathbf{n})$ and, hence, the error is bounded by $2^{-F(\mathbf{n})}$. In the case where F is a one-to-one mapping, we obtain a modulus of continuity $m(\mathbf{n}) = F^{-1}(\mathbf{n})$.

Theorem 4.2.3 ([FHP15]). *If a program $[\alpha] \rightarrow [\alpha]$ admitting a polynomial interpretation computes a real function on compact $\mathbb{K} \subseteq \mathbb{R}$, then this function is in $\text{FP}(\mathbb{R})$.*

Theorem 4.2.4 ([FHP15]). *Any polynomial-time computable real function (defined over \mathbb{K}) can be implemented by a program admitting a polynomial interpretation.*

4.2.4 A higher-order characterization of feasible functionals at any type

An alternative characterization of type-2 BFF to functions was provided in [HP17] in terms of functions over natural numbers (that can be viewed as streams/sequences of integers). For that purpose, we consider the higher-order language of Subsection 3.3.2 and the notion of interpretation introduced in Definition 3.3.3 (and Figure 3.16).

Definition 4.2.11 (Order). *Given a term M of type T , i.e. $\emptyset; \Delta \vdash M :: T$, the order of M is equal to the order of T , denoted $\text{ord}(T)$ and defined inductively by:*

$$\begin{aligned} \text{ord}(b) &= 0, & \text{if } b \in \mathbf{B}, \\ \text{ord}(T \rightarrow T') &= \max(\text{ord}(T) + 1, \text{ord}(T')) & \text{otherwise.} \end{aligned}$$

We extend Definition 4.2.3 to polynomials at any order.

Definition 4.2.12. *We consider types built from the basic type $\bar{\mathbb{N}}$ as follows:*

$$A, B ::= \bar{\mathbb{N}} \mid A \rightarrow B.$$

Higher-order polynomials are built by the following grammar:

$$P, Q ::= c^{\bar{\mathbb{N}}} \mid X^A \mid +^{\bar{\mathbb{N}} \rightarrow \bar{\mathbb{N}} \rightarrow \bar{\mathbb{N}}} \mid \times^{\bar{\mathbb{N}} \rightarrow \bar{\mathbb{N}} \rightarrow \bar{\mathbb{N}}} \mid (P^{A \rightarrow B} Q^A)^B \mid (\Lambda X^A. P^B)^{A \rightarrow B}.$$

where c represents constants in $\bar{\mathbb{N}}$ and P^A means that P is of type A . A polynomial P^A is of order i if $\text{ord}(A) = i$. When A is explicit from the context, we use the notation P_i to denote a polynomial of order i .

In the above definition, constants of type $\bar{\mathbb{N}}$ cannot be $\top \in \bar{\mathbb{N}}$. By definition, a higher-order polynomial P_i has arguments of order at most $i - 1$. For notational convenience, we will use the application of $+$ and \times with an infix notation as in the following example.

Example 4.2.4. *Here are several examples of polynomials generated by the grammar of Definition 4.2.12:*

- $P_1 = \Lambda X_0. (6 \times X_0^2 + 5)$ is an order 1 polynomial,
- $Q_1 = \times$ is an order 1 polynomial,
- $P_2 = \Lambda X_1. \Lambda X_0. (3 \times (X_1 (6 \times X_0^2 + 5)) + X_0)$ is an order 2 polynomial,
- $Q_2 = \Lambda X_1. \Lambda X_0. ((X_1 (X_1 4)) + (X_1 X_0))$ is an order 2 polynomial.

We are now ready to define the class of functions computed by terms admitting an interpretation that is bounded by higher-order polynomials.

(Procedures) $\ni P$	$::= \mathbf{Procedure} \ v^{\tau_1 \times \dots \times \tau_n \rightarrow \mathbb{D}}(v_1^{\tau_1}, \dots, v_n^{\tau_n}) \ P^* \ V \ I^* \ \mathbf{Return} \ v_r^{\mathbb{D}}$
(Declarations) $\ni V$	$::= \mathbf{Var} \ v_1^{\mathbb{D}}, \dots, v_n^{\mathbb{D}};$
(Instructions) $\ni I$	$::= v^{\mathbb{D}} := E; \mid \mathbf{Loop} \ v_0^{\mathbb{D}} \ \mathbf{with} \ v_1^{\mathbb{D}} \ \mathbf{do} \ \{I^*\}$
(Expressions) $\ni E$	$::= 1 \mid v^{\mathbb{D}} \mid v_0^{\mathbb{D}} + v_1^{\mathbb{D}} \mid v_0^{\mathbb{D}} - v_1^{\mathbb{D}} \mid v_0^{\mathbb{D}} \# v_1^{\mathbb{D}} \mid v^{\tau_1 \times \dots \times \tau_n \rightarrow \mathbb{D}}(A_1^{\tau_1}, \dots, A_n^{\tau_n})$
(Arguments) $\ni A$	$::= v \mid \lambda v_1, \dots, v_n. v(v'_1, \dots, v'_m) \quad \text{with } v \notin \{v_1, \dots, v_n\}$

Figure 4.2: BTLP grammar

Definition 4.2.13. Let \mathbf{FP}_i , $i > 0$, be the class of polynomial functionals at order i that consist in functionals computed by closed terms \mathbf{M} such that $\emptyset; \Delta \vdash \mathbf{M} :: \mathbf{T}$ over the basic type \mathbf{Nat} of unary numbers and such that:

- $\text{ord}(\mathbf{T}) = i$,
- $[\mathbf{M}]_\rho$ is bounded by an order i polynomial (i.e. $\exists P_i, [\mathbf{M}]_\rho \leq P_i$).

We will now introduce Bounded Typed Loop Programs from [IKR02] which provide an extension of the original complexity class \mathbf{BFF}_2 to any order using a programming language named BTLP.

Definition 4.2.14 (BTLP). A *Bounded Typed Loop Program* (BTLP) is a non-recursive and well-formed procedure defined by the grammar of Figure 4.2.

The well-formedness assumption is given by the following constraints: each procedure is supposed to be well-typed with respect to simple types over \mathbb{D} , the set of natural numbers of dyadic representation over $\{0, 1\}$ ($0 \equiv \epsilon$, $1 \equiv 0$, $2 \equiv 1$, $3 \equiv 00$, ...). When needed, types are explicitly mentioned in variables' superscript. Each variable of a BTLP procedure is bound by either the procedure declaration parameter list, a local variable declaration, or a lambda abstraction. In a loop statement, the guard variables v_0 and v_1 cannot be assigned to within I^* . In what follows v_1 will be called the *loop bound*.

The operational semantics of BTLP procedures is standard: parameters are passed by CBV. $+$, $-$ and $\#$ denote addition, proper subtraction, and smash function (i.e. $x \# y = 2^{|x| \times |y|}$, the size $|x|$ of the number x being the size of its dyadic representation), respectively. Each loop statement is evaluated by iterating $|v_0|$ -many times the loop body instruction under the following restriction: if an assignment $v := E$ is to be executed within the loop body, we check if the value obtained by evaluating E is of size smaller than the size of the loop bound $|v_1|$. If not then the result of evaluating this assignment is to assign 0 to v .

Definition 4.2.15 (\mathbf{BFF}_i). For any $i \geq 1$, \mathbf{BFF}_i is the class of order i functionals computable by a BTLP procedure.

It is straightforward that $\mathbf{BFF}_1 = \mathbf{FP}$ and $\mathbf{BFF}_2 = \mathbf{BFF}$.

Now we restrict the domain of \mathbf{BFF}_i classes to inputs in \mathbf{BFF}_k for $k < i$, the obtained classes are named \mathbf{SFF} for Safe Feasible Functionals.

Definition 4.2.16 (\mathbf{SFF}_i). \mathbf{SFF}_1 is defined to be the class of order 1 functionals computable by a BTLP procedure and, for any $i \geq 1$, \mathbf{SFF}_{i+1} is the class of order $i + 1$ functionals computable by

a BTLP procedure on the input domain SFF_i . In other words,

$$\begin{aligned} \text{SFF}_1 &= \text{BFF}_1, \\ \forall i \geq 1, \text{SFF}_{i+1} &= \text{BFF}_{i+1} \upharpoonright_{\text{SFF}_i} \end{aligned}$$

This is not a huge restriction since we want an arbitrary term of a given complexity class at order i to compute over terms that are already in classes of the same family at order k , for $k < i$. Consequently, programs can be built in a constructive way component by component. Another strong argument in favor of this domain restriction is that the partial evaluation of a functional at order i will, at the end, provide a function in $\mathbb{N} \rightarrow \mathbb{N}$ that is shown to be in BFF_1 ($=\text{FP}$).

We are now ready to provide a definition of Basic Feasible Functionals at any order.

Theorem 4.2.5 ([HP17]). *For any order $i \geq 1$, the class of functions in FP_i over FP_k , $k < i$, is exactly the class of functionals in SFF_i . In other words, $\text{SFF}_i \equiv \text{FP}_{i \upharpoonright (\cup_{k \leq i} \text{FP}_k)}$, for all $i \geq 1$.*

Example 4.2.5. *We have shown in Example 3.3.7, that the program*

$$\begin{aligned} \mathbf{M} := \text{letRec } f &= \lambda g. \lambda x. \text{case } x \text{ of } c \ y \ z : c \ (g \ y) \ (f \ g \ z) \\ &| \text{nil} : \text{nil} \end{aligned}$$

is such that $\text{ord}(\mathbf{M}) = 2$ and admits an interpretation bounded by $\Lambda[\mathbf{g}]_\rho. \Lambda[\mathbf{x}]_\rho. (5 \oplus ([\mathbf{g}]_\rho [\mathbf{x}]_\rho)) \times [\mathbf{x}]_\rho$. This is a second order polynomial as it can be rewritten equivalently as $\Lambda X_1. \Lambda X_0. (5 + X_1(X_0)) \times X_0$. Consequently, it belongs to FP_2 and hence computes a function of SFF_2 when fed with input in FP .

4.3 Coinductive datatypes in the light affine lambda calculus

The categorical notions of algebras and coalgebras [JR97] can provide different forms of recursion and corecursion that can be used to program algorithms in different ways. Moreover, algebras and coalgebras also provide some form of induction and coinduction that we can use to prove program properties.

Despite the fact that coalgebras correspond to infinite data types, interestingly coalgebras (and algebra) can be added to languages that are strongly normalizing while preserving the strong normalization property, as shown by [Hag87]. Moreover, algebras and coalgebras can also be encoded by using parametric polymorphism in strongly normalizing languages such as System F as shown by [Wra93].

We consider the definability of algebras and coalgebras in the Light Affine Lambda Calculus (LALC), a term language for LAL (see Subsection 1.2.4), as studied in [GP15a]. Our aim is to get a better understanding of the expressive power of LALC with respect to the definability of inductive and coinductive data structures, in particular with a focus on infinite data structures like streams. We want to define such kind of data without breaking the polynomial time normalization properties of the type system.

LALC can be seen as a subsystem of System F . However, not surprisingly, the standard System F encoding of (co)algebra cannot be straightforwardly adapted to LALC for technical reasons: variable duplication in the terms enforces the modalities $!$ and \S to appear and to propagate to the functor. Consequently, new types for encoding initial algebras and final coalgebras are required. Consider a functor F over types.

The initial algebra for F can be encoded in LALC by terms of type

$$\forall X. !(F(X) \multimap X) \multimap \S X$$

$$\begin{aligned}
 \tau, \sigma ::= & X \mid \mathbf{1} \mid !\tau \mid \S\tau \mid \tau \oplus \sigma \mid \tau \otimes \sigma \mid \tau \multimap \sigma \mid \forall X.\tau \mid \exists X.\tau \\
 \mathbf{M}, \mathbf{N}, \mathbf{L} ::= & \mathbf{x} \mid () \mid \lambda \mathbf{x} : \tau.\mathbf{M} \mid \mathbf{M}\mathbf{N} \mid \Lambda X.\mathbf{M} \mid \mathbf{M}\tau \mid !\mathbf{M} \mid \S\mathbf{M} \\
 & \mid \text{let } \S\mathbf{x} : \tau = \mathbf{M} \text{ in } \mathbf{N} \mid \text{let } !\mathbf{x} : \tau = \mathbf{M} \text{ in } \mathbf{N} \\
 & \mid \langle \mathbf{M}, \mathbf{N} \rangle \mid \text{let } \langle \mathbf{x} : \tau_1, \mathbf{y} : \tau_2 \rangle = \mathbf{M} \text{ in } \mathbf{N} \\
 & \mid \text{pack } (\mathbf{M}, \sigma) \text{ as } \tau \mid \text{unpack } \mathbf{M} \text{ as } (X, \mathbf{x}) \text{ in } \mathbf{N} \\
 & \mid \text{let } () = \mathbf{M} \text{ in } \mathbf{N} \mid \text{inj}_i^\tau(\mathbf{M}) \mid \\
 & \mid \text{case } \mathbf{M} \text{ of } \{ \text{inj}_0^\tau(\mathbf{x}) \rightarrow \mathbf{N} \mid \text{inj}_1^\tau(\mathbf{x}) \rightarrow \mathbf{L} \}
 \end{aligned}$$

Figure 4.3: Syntax of LALC

whereas they are encoded as $\forall X.(F(X) \rightarrow X) \rightarrow X$ in System F .

The final coalgebra for the same functor F can instead be encoded by terms of type

$$\exists X.!(X \multimap F(X)) \otimes \S X$$

whereas they are encoded as $\exists X.(X \rightarrow F(X)) \times X$ in System F .

Initial algebras and final coalgebras definable in System F are only *weak* (i.e. existence but no uniqueness). In the case of LALC the two types above provide even more restricted classes of initial algebras and final coalgebras: the ones that enjoy some distributive properties with the \S modality. More precisely, for initial algebras we need functors that *left-distribute* over \S , i.e. functors F such that $F(\S X) \multimap \S F(X)$. Conversely, for final algebras we need functors that *right-distribute* over \S , i.e. functors F such that $\S F(X) \multimap F(\S X)$.

Functors that left-distribute over \S are quite common in LALC and so we can define several standard inductive data types. Unfortunately, only few functors right-distribute over \S . In particular, we cannot encode standard coinductive data structures. The main reason is that the modality \S does not *distribute* with respect to the connectives tensor and plus. More precisely, in LALC we cannot derive the distribution laws $\S(A \otimes B) \multimap \S A \otimes \S B$ and $\S(A \oplus B) \multimap \S A \oplus \S B$ for generic A and B . We overcome this situation by adding terms for these distributive laws to the language LALC. Thanks to this extensions we are able to write programs working on infinite streams of Boolean numbers (or of any finite data type) and other infinite data types. A more complete and detailed discussion about the soundness of this extension can be found in [GP15a].

4.3.1 Light affine lambda calculus

The syntax of the Light Affine Lambda Calculus (LALC) is inspired by the type restrictions of LAL that we have described in Subsection 1.2.4. The types and the terms of LALC are presented in Figure 4.3. The multiplicative unit $\mathbf{1}$ is the only basic type. The type constructors consist of the linear implication \multimap , the tensor product \otimes , and the additive disjunction \oplus . There are type variables X , a universal quantifier $\forall X.\tau$, and an existential quantifier $\exists X.\tau$. As in LAL, we also have the two modalities $!$ and \S . Every type constructor comes equipped with a term constructor and a term destructor.

The semantics of LALC is defined in terms of the reduction relation \rightarrow described in Figure 4.4 where we use the notations $[\mathbf{M}/\mathbf{x}]$ and $[\tau/X]$ for the usual capture avoiding substitution on terms and on types, respectively. Let \dagger, \ddagger be modality meta-variables denoting the modalities $!$ or \S . In Figure 4.4, we have omitted several commuting rules. The number of these rules is quite high and their behavior is standard. We provide only the last two rules as representative of this class.

$$\begin{aligned}
 & (\lambda \mathbf{x} : \tau. \mathbf{M}) \mathbf{N} \rightarrow \mathbf{M}[\mathbf{N}/\mathbf{x}] \\
 & \text{let } () = () \text{ in } \mathbf{M} \rightarrow \mathbf{M} \\
 & (\Lambda X. \mathbf{M}) \tau \rightarrow \mathbf{M}[\tau/X] \\
 \text{case } \text{inj}_i^\tau(\mathbf{M}) \text{ of } \{ \text{inj}_0^\tau(\mathbf{x}) \rightarrow \mathbf{N}_0 \mid \text{inj}_1^\tau(\mathbf{x}) \rightarrow \mathbf{N}_1 \} & \rightarrow \mathbf{N}_i[\mathbf{M}/\mathbf{x}] \\
 & \text{let } \langle \mathbf{x} : \tau, \mathbf{y} : \sigma \rangle = \langle \mathbf{N}_0, \mathbf{N}_1 \rangle \text{ in } \mathbf{M} \rightarrow \mathbf{M}[\mathbf{N}_0/\mathbf{x}, \mathbf{N}_1/\mathbf{y}] \\
 \text{unpack } (\text{pack } (\mathbf{M}, \sigma) \text{ as } \exists X. \tau) \text{ as } (X, \mathbf{x}) \text{ in } \mathbf{N} & \rightarrow \mathbf{N}[\mathbf{M}/\mathbf{x}, \sigma/X] \\
 & \text{let } \S \mathbf{x} : \S \tau = \S \mathbf{N} \text{ in } \mathbf{M} \rightarrow \mathbf{M}[\mathbf{N}/\mathbf{x}] \\
 & \text{let } !\mathbf{x} : !\tau = !\mathbf{N} \text{ in } \mathbf{M} \rightarrow \mathbf{M}[\mathbf{N}/\mathbf{x}] \\
 & (\text{let } \dagger \mathbf{x} : \dagger \tau = \mathbf{N} \text{ in } \mathbf{M}) \mathbf{L} \rightarrow \text{let } \dagger \mathbf{x} : \dagger \tau = \mathbf{N} \text{ in } (\mathbf{M}\mathbf{L}) \\
 \text{let } \dagger \mathbf{y} : \dagger \tau = (\text{let } \dagger \mathbf{x} : \dagger \sigma = \mathbf{N} \text{ in } \mathbf{L}) \text{ in } \mathbf{M} & \rightarrow \text{let } \dagger \mathbf{x} : \dagger \sigma = \mathbf{N} \text{ in } (\text{let } \dagger \mathbf{y} : \dagger \tau = \mathbf{L} \text{ in } \mathbf{M})
 \end{aligned}$$

Figure 4.4: Semantics of LALC

Let an environment Γ be a map from types to term variables. A typing judgment is of the shape $\Gamma \vdash \mathbf{M} : \tau$, for some typing environment Γ , some term \mathbf{M} and some type τ . The standard typing rules, inherited from LAL, together with the rules for the extra constructs are given in Figure 4.5. As usual, this system uses the notion of discharged formulas, which are expressions of the form $[\tau]_{\dagger}$. Given a typing environment $\Gamma = \mathbf{x}_1 : \tau_1, \dots, \mathbf{x}_n : \tau_n$, $[\Gamma]_{\dagger}$ is a notation for the environment $\mathbf{x}_1 : [\tau_1]_{\dagger}, \dots, \mathbf{x}_n : [\tau_n]_{\dagger}$.

We can characterize FP using LALC. This characterization is similar to the one of LAL presented in Theorem 1.2.3. Let the *depth* $d(\mathbf{M})$ of a term \mathbf{M} : the maximal number of nested $!$ or \S that can be found in any path of the term syntax tree. Let the data type \mathbb{B}^* be a standard Church encoding of strings of Boolean numbers.

Theorem 4.3.1 (FP soundness and completeness). *Consider a term $\Gamma \vdash \mathbf{M} : \tau$. Then, \mathbf{M} can be reduced to normal form by a TM working in time polynomial in $|\mathbf{M}|$ with exponent proportional to $d(\mathbf{M})$. Conversely, for every function $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ in FP there exists a natural number n and a term $\mathbf{M} : \mathbb{B}^* \multimap \S^n \mathbb{B}^*$ such that \mathbf{M} computes f .*

4.3.2 Algebra and coalgebra in System F

Let us start by recalling the standard notions of F -algebras and F -coalgebras.

Definition 4.3.1 (F -Algebra and F -Coalgebra). *Given a category \mathcal{C} and a (endo)functor $F : \mathcal{C} \rightarrow \mathcal{C}$:*

- a F -algebra is pair (A, a) of an object $A \in \mathcal{C}$ together with a \mathcal{C} -morphism $a : F(A) \rightarrow A$,
- a F -coalgebra is pair (A, a) of an object $A \in \mathcal{C}$ together with a \mathcal{C} -morphism $a : A \rightarrow F(A)$.

We can define two categories $\text{Alg-}F$ and $\text{Coalg-}F$ whose objects are F -algebras and F -coalgebras, respectively, and whose morphisms are defined as follows.

Definition 4.3.2. *A F -algebra homomorphism from the F -algebra (A, a) to the F -algebra (B, b)*

$$\begin{array}{c}
\frac{}{\mathbf{x} : \tau \vdash \mathbf{x} : \tau} \text{ (Ax)} \quad \frac{\Gamma \vdash \mathbf{M} : \tau}{\Gamma, \Delta \vdash \mathbf{M} : \tau} \text{ (W)} \quad \frac{\Gamma, \mathbf{x} : [\tau]!, \mathbf{y} : [\tau]! \vdash \mathbf{M} : \sigma}{\Gamma, \mathbf{z} : [\tau]! \vdash \mathbf{M}[\mathbf{z}/\mathbf{x}, \mathbf{z}/\mathbf{y}] : \sigma} \text{ (C)} \\
\\
\frac{\Gamma, \mathbf{x} : \tau \vdash \mathbf{M} : \sigma}{\Gamma \vdash \lambda \mathbf{x} : \tau. \mathbf{M} : \tau \multimap \sigma} \text{ (}\multimap\text{I)} \quad \frac{\Gamma \vdash \mathbf{M} : \tau \multimap \sigma \quad \Delta \vdash \mathbf{N} : \tau}{\Gamma, \Delta \vdash \mathbf{M}\mathbf{N} : \sigma} \text{ (}\multimap\text{E)} \\
\\
\frac{\Gamma, \Delta \vdash \mathbf{M} : \tau}{[\Gamma]!, [\Delta]_{\S} \vdash \S \mathbf{M} : \S \tau} \text{ (§I)} \quad \frac{\Gamma \vdash \mathbf{N} : \S \tau \quad \Delta, \mathbf{x} : [\tau]_{\S} \vdash \mathbf{M} : \sigma}{\Gamma, \Delta \vdash \text{let } \S \mathbf{x} : \S \tau = \mathbf{N} \text{ in } \mathbf{M} : \sigma} \text{ (§E)} \\
\\
\frac{\Gamma \vdash \mathbf{M} : \tau \quad \Gamma \subseteq \{\mathbf{x} : \sigma\}}{[\Gamma]! \vdash !\mathbf{M} : !\tau} \text{ (!I)} \quad \frac{\Gamma \vdash \mathbf{N} : !\tau \quad \Delta, \mathbf{x} : [\tau]! \vdash \mathbf{M} : \sigma}{\Gamma, \Delta \vdash \text{let } !\mathbf{x} : !\tau = \mathbf{N} \text{ in } \mathbf{M} : \sigma} \text{ (!E)} \\
\\
\frac{\Gamma \vdash \mathbf{M} : \tau \quad X \notin FV(\Gamma)}{\Gamma \vdash \Lambda X. \mathbf{M} : \forall X. \tau} \text{ (\forall I)} \quad \frac{\Gamma \vdash \mathbf{M} : \forall X. \tau}{\Gamma \vdash \mathbf{M} \sigma : \tau[\sigma/X]} \text{ (\forall E)} \\
\\
\frac{\Gamma \vdash \mathbf{M} : \tau \quad \Delta \vdash \mathbf{N} : \sigma}{\Gamma, \Delta \vdash \langle \mathbf{M}, \mathbf{N} \rangle : \tau \otimes \sigma} \text{ (\otimes I)} \quad \frac{\Gamma \vdash \mathbf{M} : \tau \otimes \sigma \quad \Delta, \mathbf{x} : \tau, \mathbf{y} : \sigma \vdash \mathbf{N} : \tau'}{\Gamma, \Delta \vdash \text{let } \langle \mathbf{x} : \tau, \mathbf{y} : \sigma \rangle = \mathbf{M} \text{ in } \mathbf{N} : \tau'} \text{ (\otimes E)} \\
\\
\frac{}{\Gamma \vdash () : \mathbf{1}} \text{ (1I)} \quad \frac{\Gamma \vdash \mathbf{M} : \mathbf{1} \quad \Delta \vdash \mathbf{N} : \tau}{\Gamma, \Delta \vdash \text{let } () = \mathbf{M} \text{ in } \mathbf{N} : \tau} \text{ (1E)} \\
\\
\frac{\Gamma \vdash \mathbf{M} : \tau_i}{\Gamma \vdash \text{inj}_i^{\tau_0 \oplus \tau_1}(\mathbf{M}) : \tau_0 \oplus \tau_1} \text{ (\oplus I)} \quad \frac{\Gamma \vdash \mathbf{M} : \tau_0 \oplus \tau_1 \quad \forall i, \Delta, \mathbf{x} : \tau_i \vdash \mathbf{N}_i : \tau}{\Gamma, \Delta \vdash \text{case } \mathbf{M} \text{ of } \{\text{inj}_0^{\tau_0 \oplus \tau_1}(\mathbf{x}) \rightarrow \mathbf{N}_0 \mid \text{inj}_1^{\tau_0 \oplus \tau_1}(\mathbf{x}) \rightarrow \mathbf{N}_1\} : \tau} \text{ (\oplus E)} \\
\\
\frac{\Gamma \vdash \mathbf{M} : \tau[\sigma/X]}{\Gamma \vdash \text{pack } (\mathbf{M}, \sigma) \text{ as } \exists X. \tau : \exists X. \tau} \text{ (\exists I)} \quad \frac{\Gamma \vdash \mathbf{M} : \exists X. \tau \quad \Delta, \mathbf{x} : \tau \vdash \mathbf{N} : \sigma}{\Gamma, \Delta \vdash \text{unpack } \mathbf{M} \text{ as } (\mathbf{X}, \mathbf{x}) \text{ in } \mathbf{N} : \sigma} \text{ (\exists E)}
\end{array}$$

Figure 4.5: Type system for LALC

is a morphism $f : A \rightarrow B$ making the following diagram commute:

$$\begin{array}{ccc} F(A) & \xrightarrow{F(f)} & F(B) \\ \downarrow a & & \downarrow b \\ A & \xrightarrow{f} & B \end{array}$$

A F -coalgebra homomorphism from the F -coalgebra (A, a) to the F -coalgebra (B, b) is a morphism $f : A \rightarrow B$ making the following diagram commute:

$$\begin{array}{ccc} A & \xrightarrow{f} & B \\ \downarrow a & & \downarrow b \\ F(A) & \xrightarrow{F(f)} & F(B) \end{array}$$

To define the traditional inductive and coinductive data types we also need the notions of *initial algebras* and *final coalgebras*.

Definition 4.3.3 (Initial algebra and final coalgebra). A F -algebra (A, a) is *initial* if for each F -algebra (B, b) , there exists a unique F -algebra homomorphism $f : A \rightarrow B$. A F -coalgebra (A, a) is *final* if for each F -coalgebra (B, b) , there exists a unique F -coalgebra homomorphism $f : B \rightarrow A$.

If the uniqueness condition is not met then the F -algebra (F -coalgebra, respectively) is only weakly initial (weakly final, respectively).

An initial F -algebra is an initial object in the category $\mathbf{Alg}\text{-}F$. Conversely, a final F -coalgebra is a terminal object in the category $\mathbf{Coalg}\text{-}F$.

A functor $F(X)$ is definable in System F if $F(X)$ is a type scheme mapping every type A to the type $F(A)$, and if there exists a term F mapping every term of type $A \rightarrow B$ to a term of type $F(A) \rightarrow F(B)$ and such that it preserves identity and composition. We say that a functor $F(X)$ is covariant if the variable X only appears in covariant positions.

It is a well known result that for any covariant functor $F(X)$ that is definable in System F we can define an algebra that is weakly initial and a coalgebra that is weakly final [JR97].

Proposition 4.3.1 (Weakly Initial Algebra). Let $F(X)$ be a covariant functor definable in System F and $\mathbf{T} = \forall X.(F(X) \rightarrow X) \rightarrow X$. Consider the morphisms defined by:

$$\begin{aligned} \mathbf{in}_{\mathbf{T}} &: F(\mathbf{T}) \rightarrow \mathbf{T}, \\ \mathbf{in}_{\mathbf{T}} &= \lambda \mathbf{s} : F(\mathbf{T}). \Lambda X. \lambda \mathbf{k} : F(X) \rightarrow X. \mathbf{k}(F(\mathbf{fold}_{\mathbf{T}} X \mathbf{k}) \mathbf{s}), \\ \mathbf{fold}_{\mathbf{T}} &: \forall X.(F(X) \rightarrow X) \rightarrow \mathbf{T} \rightarrow X, \\ \mathbf{fold}_{\mathbf{T}} &= \Lambda X. \lambda \mathbf{k} : F(X) \rightarrow X. \lambda \mathbf{t} : \mathbf{T}. \mathbf{t} X \mathbf{k}. \end{aligned}$$

Then, $(\mathbf{T}, \mathbf{in}_{\mathbf{T}})$ is a weakly initial F -algebra: for every F -algebra $(A, g : F(A) \rightarrow A)$ there is a F -homomorphism $h : \mathbf{T} \rightarrow A$ defined as $h = \mathbf{fold}_{\mathbf{T}} A g$.

We will sometimes write T as $\mu X.F(X)$ when we want to stress the underlying functor F and the fact that T corresponds to the least fixpoint of F .

Example 4.3.1 ([GP15a]). Let us consider a functor defined on types as $F(X) = \mathbf{1} + X$ and on terms as:

$$\lambda f : X \rightarrow Y. \lambda x : \mathbf{1} + X. \text{case } x \text{ of } \{ \text{inj}_0^{1+X}(z) \rightarrow \text{inj}_0^{1+Y}(()), \text{inj}_1^{1+X}(z) \rightarrow \text{inj}_1^{1+Y}(f z) \}.$$

Let $\mathbb{N} = \mu X.F(X)$. Proposition 4.3.1 ensures that $(\mathbb{N}, \text{in}_{\mathbb{N}})$ is a weak initial algebra: the weakly initial algebra of natural numbers. In particular, we can define $\underline{0} = \text{in}_{\mathbb{N}}(\text{inj}_0^{1+\mathbb{N}}(()))$, $n + \underline{1} = \text{in}_{\mathbb{N}}(\text{inj}_1^{1+\mathbb{N}}(n))$, and more in general the successor function as $\text{succ} = \lambda x. \text{in}_{\mathbb{N}}(\text{inj}_1^{1+\mathbb{N}}(x))$. We can use the fact that \mathbb{N} is a weakly initial algebra to define an addition function. We just need to consider a term like the following (we omit some type for conciseness):

$$g = \lambda x : \mathbf{1} + (\mathbb{N} \rightarrow \mathbb{N}). \text{case } x \text{ of } \{ \text{inj}_0(z) \rightarrow \lambda y : \mathbb{N}. y, \text{inj}_1(z) \rightarrow \lambda y : \mathbb{N}. \text{succ}(z y) \}.$$

Then, Proposition 4.3.1 ensures that we can define add as $\text{fold}_{\mathbb{N}}(\mathbb{N} \rightarrow \mathbb{N}) g$.

Proposition 4.3.2 (Weakly Final Coalgebra). Let F be a covariant functor definable in System F and $T = \exists X.(X \rightarrow F(X)) \times X$. Consider the morphisms defined by:

$$\begin{aligned} \text{out}_T : T &\rightarrow F(T), \\ \text{out}_T = \lambda t : T. \text{unpack } t \text{ as } (X, z) \text{ in} \\ &\quad \text{let } (k, x) = z \text{ in } F(\text{unfold}_T X k)(k x), \\ \text{unfold}_T : \forall X.(X \rightarrow F(X)) &\rightarrow X \rightarrow T, \\ \text{unfold}_T = \Lambda X. \lambda k : X \rightarrow F(X). \lambda x : X. \text{pack } ((k, x), X) &\text{ as } T. \end{aligned}$$

Then, (T, out_T) is a weakly final F -coalgebra: for every F -coalgebra $(A, g : A \rightarrow F(A))$ there is a F -homomorphism $h : A \rightarrow T$ defined as $h = \text{unfold}_T A g$.

Similarly to the case of F -algebras, we will write T as $\nu X.F(X)$ when we want to stress the underlying functor F and the fact that T corresponds to the greatest fixpoint of F .

Example 4.3.2 ([GP15a]). Let us consider a functor defined on types as $F(X) = \mathbb{N} \times X$ and on terms as:

$$\lambda f : X \rightarrow Y. \lambda x : \mathbb{N} \times X. \text{let } \langle x_1, x_2 \rangle = x \text{ in } \langle x_1, f x_2 \rangle.$$

Let $\mathbb{N}^\omega = \nu X.F(X)$. Proposition 4.3.2 ensures that $(\mathbb{N}^\omega, \text{out}_{\mathbb{N}^\omega})$ is a weak final coalgebra: the weakly final coalgebra of streams over natural numbers. We can define the usual operations on streams as $\text{head} = \lambda x : \mathbb{N}^\omega. \text{let } \langle x_1, x_2 \rangle = (\text{out}_{\mathbb{N}^\omega} x) \text{ in } x_1$, and $\text{tail} = \lambda x : \mathbb{N}^\omega. \text{let } \langle x_1, x_2 \rangle = (\text{out}_{\mathbb{N}^\omega} x) \text{ in } x_2$. We can use the fact that \mathbb{N}^ω is a weakly final coalgebra to define streams. As an example we can define a constant stream of k s by using a function:

$$g = \lambda x : \mathbf{1}. \text{let } () = x \text{ in } \langle k, () \rangle.$$

Proposition 4.3.2 ensures that we can define $\text{const} = \text{unfold}_{\mathbb{N}^\omega} \mathbf{1} g()$. Similarly, we can define a function that extracts from a stream the elements in even position. This time we need a function:

$$g = \lambda x : \mathbb{N}^\omega. \langle \text{hd } x, \text{tl } (\text{tl } x) \rangle.$$

Proposition 4.3.2 ensures that we can define $\text{even} = \text{unfold}_{\mathbb{N}^\omega} \mathbb{N}^\omega g$.

4.3.3 Algebra in the light affine lambda calculus

Now we try to adapt the notions of algebra and coalgebra to the LALC setting. As already mentioned, the candidate type for weakly initial algebra is:

$$\mathsf{T} = \forall X.!(F(X) \multimap X) \multimap \S X.$$

However two restrictions are needed. First, the modality \S in the above equation propagates when one wants to build the F -homomorphism to another F -algebra. Consequently, only weakly initial algebras of the shape $(\S B, g : F(\S B) \rightarrow \S B)$ can be considered. Second, the functor F has to *left-distribute* over \S . This corresponds to require the existence of a morphism:

$$L_F : F(\S X) \multimap \S F(X).$$

Putting these two restrictions together we can formulate an alternative definition of weakly initial algebra as follows.

Definition 4.3.4. *Given a functor F , we say that an F -algebra (A, a) is weakly initial under \S if for every F -algebra of the form (B, f) there exists an F -algebra $(\S B, g)$ and an F -algebra homomorphism $h : A \rightarrow \S B$ making the following diagram commute:*

$$\begin{array}{ccc}
 F(A) & \xrightarrow{F(h)} & F(\S B) \\
 \downarrow a & & \downarrow g \\
 A & \xrightarrow{h} & \S B
 \end{array}
 \begin{array}{c}
 \\
 \\
 \\
 \end{array}
 \begin{array}{ccc}
 & & \S F(B) \\
 & \swarrow L_F & \\
 & & \\
 & \searrow \S f & \\
 & &
 \end{array}$$

Theorem 4.3.2 ([GP15a]). *Let F be a functor definable in LALC that left-distributes over \S , and let $\mathsf{T} = \forall X.!(F(X) \multimap X) \multimap \S X$. Consider the morphisms defined by:*

$$\begin{aligned}
 \mathsf{in}_{\mathsf{T}} &: F(\mathsf{T}) \multimap \mathsf{T}, \\
 \mathsf{in}_{\mathsf{T}} &= \lambda \mathsf{s} : F(\mathsf{T}). \Lambda X. \lambda \mathsf{k} : !(F(X) \multimap X). \mathsf{let} \ !\mathsf{y} : !(F(X) \multimap X) = \mathsf{k} \ \mathsf{in} \\
 &\quad \mathsf{let} \ \S \mathsf{z} : \S F(X) = L_F(F(\mathsf{fold}_{\mathsf{T}} X \ !\mathsf{y}) \ \mathsf{s}) \ \mathsf{in} \ \S(\mathsf{y} \ \mathsf{z}), \\
 \mathsf{fold}_{\mathsf{T}} &: \forall X.!(F(X) \multimap X) \multimap \mathsf{T} \multimap \S X, \\
 \mathsf{fold}_{\mathsf{T}} &= \Lambda X. \lambda \mathsf{k} : !(F(X) \multimap X). \lambda \mathsf{p} : \mathsf{T}. \mathsf{p} \ X \ \mathsf{k}.
 \end{aligned}$$

Then, $(\mathsf{T}, \mathsf{in}_{\mathsf{T}})$ is a weakly initial F -algebra under \S : for every F -algebra $(B, f : F(B) \multimap B)$ we have an F -algebra $(\S B, g : F(\S B) \rightarrow \S B)$ and an F -algebra homomorphism $h : \mathsf{T} \rightarrow \S B$ defined as $h = \mathsf{fold}_{\mathsf{T}} B \ !f$.

Moreover we can give a characterization of a large class of left-distributing functors.

Lemma 4.3.1. *All the functors built using the following signature left-distribute over \S :*

$$F(X) ::= \mathbf{1} \mid X \mid A \mid \S F(X) \mid F(X) \oplus F(X) \mid F(X) \otimes F(X),$$

provided that A is a closed type for which it exists a closed term of type $A \multimap !A$ or type $A \multimap \S A$.

Example 4.3.3. Consider the functor $F(X) = \mathbf{1} \oplus X$. This is the linear analogous of the functor considered in Example 4.3.1, definable by the same term (in the types annotation, implication is replaced by a linear arrow and $+$ is replaced by \oplus). By Lemma 4.3.1, we have that F left-distribute over \S , and so by Theorem 4.3.2 we have that $(\mathbb{N}, \text{in}_{\mathbb{N}})$ is a weakly initial F -algebra under \S , where by abuse of notation we again use \mathbb{N} to denote $\mu X.F(X)$. Similarly to what we did in Example 4.3.1, we can define natural numbers as inhabitants of this type. Noticing that to the term g defined there we can also give the type $F(\mathbb{N} \multimap \mathbb{N}) \multimap (\mathbb{N} \multimap \mathbb{N})$, we have that $\text{add} = \text{fold}_{\mathbb{N}}(\mathbb{N} \multimap \mathbb{N})!g$ has type $\mathbb{N} \multimap \S(\mathbb{N} \multimap \mathbb{N})$.

4.3.4 Coalgebra in the light affine lambda calculus

To encode final coalgebra in our setting we consider the following type:

$$\mathbb{T} = \exists X.!(X \multimap F(X)) \otimes \S X.$$

By duality, we encounter problems similar to the ones encountered for the algebra encoding: we need to restrict our study to coalgebra of the shape $(\S B, g : \S B \multimap F(\S B))$. Moreover, we also need the functor F to right-distribute over \S . This corresponds to require for the existence of the following morphism:

$$R_F : \S F(X) \multimap F(\S X).$$

Definition 4.3.5. Given a functor F , we say that an F -coalgebra (A, a) is weakly final under \S if for every F -coalgebra of the form (B, f) there exists an F -coalgebra $(\S B, g)$ and an F -coalgebra homomorphism $h : \S B \rightarrow A$ making the following diagram commute:

$$\begin{array}{ccc}
 A & \xleftarrow{h} & \S B \\
 \downarrow a & & \downarrow g \\
 F(A) & \xleftarrow{F(h)} & F(\S B)
 \end{array}
 \quad
 \begin{array}{ccc}
 & & \searrow \S f \\
 & & \S F(B) \\
 & \swarrow R_F & \\
 & & F(\S B)
 \end{array}$$

Theorem 4.3.3 ([GP15a]). Let F be a functor definable in LALC that right-distribute over \S , and let $\mathbb{T} = \exists X.!(X \multimap F(X)) \otimes \S X$. Consider the morphisms defined by:

$$\begin{aligned}
 \text{out}_{\mathbb{T}} &: \mathbb{T} \multimap F(\mathbb{T}), \\
 \text{out}_{\mathbb{T}} &= \lambda t : \mathbb{T}.\text{unpack } t \text{ as } (X, z) \text{ in let } \langle k :!(X \multimap F(X)), x : \S X \rangle = z \text{ in} \\
 &\quad \text{let } !u = k \text{ in let } \S v = x \text{ in } F(\text{unfold}_{\mathbb{T}} X !u) R_F(\S(u v)), \\
 \text{unfold}_{\mathbb{T}} &: \forall X.!(X \multimap F(X)) \multimap \S X \multimap \mathbb{T}, \\
 \text{unfold}_{\mathbb{T}} &= \Lambda X.\lambda k :!(X \multimap F(X)).\lambda x : \S X.\text{pack}(\langle k, x \rangle, X) \text{ as } \mathbb{T}.
 \end{aligned}$$

Then, $(\mathbb{T}, \text{out}_{\mathbb{T}})$ is a weakly final F -coalgebra under \S : for every F -coalgebra $(B, f : B \multimap F(B))$ we have an F -coalgebra $(\S B, g : \S B \multimap F(\S B))$ and an F -coalgebra homomorphism $h : \S B \rightarrow \mathbb{T}$ defined as $h = \text{unfold}_{\mathbb{T}} B!f$.

Sadly, the coalgebra counterpart of Lemma 4.3.1 has a poor expressive power:

Lemma 4.3.2. All the functors built using the following signature right-distribute over \S :

$$F(X) ::= \mathbf{1} \mid X \mid A \mid \S F(X),$$

provided that A is a closed type for which it exists a closed term of type $\S A \multimap A$.

$\begin{aligned} \mathbf{M}, \mathbf{N}, ::= & \dots \\ & \text{dist } \S \langle \mathbf{x} : \tau_1, \mathbf{y} : \tau_2 \rangle = \mathbf{M} \text{ as } \mathbf{z} = \langle \S \mathbf{x}, \S \mathbf{y} \rangle \text{ in } \mathbf{N} \\ & \text{dist } \S \text{inj}_i^{\tau \oplus \tau'}(\mathbf{x}) = \mathbf{M} \text{ as } \mathbf{z} = \text{inj}_i^{\S \tau \oplus \S \tau'}(\S \mathbf{x}) \text{ in } \mathbf{N}. \end{aligned}$ <p style="text-align: center;">(a) Syntax of LALC distributions</p>
<hr/> $\begin{aligned} \text{dist } \S \langle \mathbf{x} : \tau, \mathbf{y} : \sigma \rangle = \S \langle \mathbf{M}_1, \mathbf{M}_2 \rangle \text{ as } \mathbf{z} = \langle \S \mathbf{x}, \S \mathbf{y} \rangle \text{ in } \mathbf{N} & \rightarrow \mathbf{N}[\langle \S \mathbf{M}_1, \S \mathbf{M}_2 \rangle / \mathbf{z}] & (\text{dis-1}) \\ \text{dist } \S \text{inj}_i^{\tau \oplus \tau'}(\mathbf{x}) = \S \text{inj}_i^{\tau \oplus \tau'}(\mathbf{M}) \text{ as } \mathbf{z} = \text{inj}_i^{\S \tau \oplus \S \tau'}(\S \mathbf{x}) \text{ in } \mathbf{N} & \rightarrow \mathbf{N}[\text{inj}_i^{\S \tau \oplus \S \tau'}(\S \mathbf{M}) / \mathbf{z}] & (\text{dis-2}) \end{aligned}$ <p style="text-align: center;">(b) Semantics of LALC distributions</p> <hr/>
$\frac{\Gamma \vdash \mathbf{M} : \S(\tau \otimes \sigma) \quad \Delta, \mathbf{z} : \S \tau \otimes \S \sigma \vdash \mathbf{N} : \tau'}{\Gamma, \Delta \vdash \text{dist } \S \langle \mathbf{x} : \tau, \mathbf{y} : \sigma \rangle = \mathbf{M} \text{ as } \mathbf{z} = \langle \S \mathbf{x}, \S \mathbf{y} \rangle \text{ in } \mathbf{N} : \tau'} \quad (\mathbf{d} \otimes)$ $\frac{\Gamma \vdash \mathbf{M} : \S(\tau \oplus \sigma) \quad \Delta, \mathbf{z} : \S \tau \oplus \S \sigma \vdash \mathbf{N} : \tau'}{\Gamma, \Delta \vdash \text{dist } \S \text{inj}_i^{\tau \oplus \sigma}(\mathbf{x}) = \mathbf{M} \text{ as } \mathbf{z} = \text{inj}_i^{\S \tau \oplus \S \sigma}(\S \mathbf{x}) \text{ in } \mathbf{N} : \tau'} \quad (\mathbf{d} \oplus)$ <p style="text-align: center;">(c) Typing rules for LALC distributions</p> <hr/>

Figure 4.6: LALC extended with distributions

This issue is solved in [GP15a] by extending the syntax with explicit distribution constructs. The extended syntax, semantics, and typing rules are provided in Figure 4.6.

Now Lemma 4.3.2 can be extended as follows.

Lemma 4.3.3. *In LALC extended with distributions, all the functors built using the following signature righ-distribute over \S :*

$$F(X) ::= \mathbf{1} \mid X \mid A \mid \S F(X) \mid F(X) \otimes F(X) \mid F(X) \oplus F(X),$$

provided that A is a closed type for which it exists a closed term of type $\S A \multimap A$.

Now an open issue solved in [GP15a] was to know whether the soundness of Theorem 4.3.1 remains true in the extended language. The main idea is that the stratification principle observed in LAL or LACL remains valid in the extended framework.

Theorem 4.3.4 (Polynomial Time Soundness of LALC extended with distributions [GP15a]). *Consider a type derivation $\Gamma \vdash \mathbf{M} : \tau$ of a term \mathbf{M} of LALC extended with distributions. Then, \mathbf{M} can be reduced to normal form by a TM working in time polynomial in $|\mathbf{M}|$ with exponent proportional to $d(\mathbf{M})$.*

Example 4.3.4 ([GP15a]). *We would like to consider streams of natural numbers as in Example 4.3.2. Unfortunately, Lemma 4.3.3 is not enough to show that the functor $F(X) = \mathbb{N} \otimes X$ distributes to the right, as the coercion $\S \mathbb{N} \multimap \mathbb{N}$ does not hold. Nevertheless, we can consider*

streams of every finite type of the shape $\mathbf{1} \oplus \dots \oplus \mathbf{1}$, including Boolean numbers. Let us consider the functor defined on types as $F(X) = \mathbb{B}_2 \otimes X$ and on terms as:

$$\lambda f : X \multimap Y. \lambda x : \mathbb{B}_2 \otimes X. \text{let } \langle x_1, x_2 \rangle = x \text{ in } \langle x_1, f x_2 \rangle.$$

This functor right-distributes by Lemma 4.3.3.

Let $\mathbb{B}_2^\omega = \nu X. F(X)$. Theorem 4.3.3 ensures that $(\mathbb{B}_2^\omega, \text{out}_{\mathbb{B}_2^\omega})$ is a weak final coalgebra. We can define the constant stream of 1 (as a Boolean number) as follows $\text{ones} = \text{unfold } \mathbf{1}!(\lambda x : \mathbf{1}. \text{let } () = x \text{ in } \langle \mathbf{1}, () \rangle) \S()$.

As in System F , we can define the usual operations on streams as:

$$\begin{aligned} \text{head} &= \lambda x : \mathbb{B}_2^\omega. \text{let } \langle x_1, x_2 \rangle = (\text{out}_{\mathbb{B}_2^\omega} x) \text{ in } x_1, \\ \text{tail} &= \lambda x : \mathbb{B}_2^\omega. \text{let } \langle x_1, x_2 \rangle = (\text{out}_{\mathbb{B}_2^\omega} x) \text{ in } x_2. \end{aligned}$$

Unfortunately, using these operations is often inconvenient in presence of linearity, and it is more convenient to use directly the coalgebra structure provided by $\text{out}_{\mathbb{B}_2^\omega}$. Consider for example the operation that extracts from a stream of Boolean numbers the elements in even position – we have seen a similar operation encoded in System F in Example 4.3.2. We can define this operation by using the term:

$$\mathbf{M} = \lambda x : \mathbb{B}_2^\omega. \text{let } \langle x_1, x_2 \rangle = (\text{out}_{\mathbb{B}_2^\omega} x) \text{ in let } \langle x_{21}, x_{22} \rangle = (\text{out}_{\mathbb{B}_2^\omega} x_2) \text{ in } \langle x_1, x_{22} \rangle.$$

\mathbf{M} has type $\mathbb{B}_2^\omega \multimap \mathbb{B}_2 \otimes \mathbb{B}_2^\omega$. So, by Theorem 4.3.3 $\text{even} = \text{unfold}_{\mathbb{B}_2^\omega} \mathbb{B}_2^\omega !\mathbf{M}$. Another interesting example is the term computing the merge of two streams:

$$\mathbf{M} = \lambda x : \mathbb{B}_2^\omega \otimes \mathbb{B}_2^\omega. \text{let } \langle x_1, x_2 \rangle = x \text{ in let } \langle x_{11}, x_{12} \rangle = (\text{out}_{\mathbb{B}_2^\omega} x_1) \text{ in } \langle x_{11}, \langle x_2, x_{12} \rangle \rangle.$$

\mathbf{M} has type $\mathbb{B}_2^\omega \otimes \mathbb{B}_2^\omega \multimap \mathbb{B}_2 \otimes (\mathbb{B}_2^\omega \otimes \mathbb{B}_2^\omega)$. So, by Theorem 4.3.3 again, $\text{merge} = \text{unfold}_{\mathbb{B}_2^\omega} (\mathbb{B}_2^\omega \otimes \mathbb{B}_2^\omega) !\mathbf{M}$.

We can combine algebra and coalgebra examples. For example, consider the inductive function take that for a given n returns the first n elements of a stream as a string:

$$\begin{aligned} \% &= \lambda x : \mathbf{1} \oplus (\mathbb{B}_2^\omega \multimap \mathbb{B}_2^*). \text{case } x \text{ of} \\ &\{ \text{inj}_0(z) \rightarrow \lambda y : \mathbb{B}_2^\omega. \text{nil} \mid \text{inj}_1(z) \rightarrow \lambda y : \mathbb{B}_2^\omega. \text{let } \langle y_1, y_2 \rangle = (\text{out}_{\mathbb{B}_2^\omega} y) \text{ in } \text{cons}(y_1, z y_2) \}. \end{aligned}$$

The term \mathbf{M} has type $\mathbf{1} \oplus (\mathbb{B}_2^\omega \multimap \mathbb{B}_2^*) \multimap (\mathbb{B}_2^\omega \multimap \mathbb{B}_2^*)$. Consequently, by Theorem 4.3.2, $\text{take} = \text{fold}_{\mathbb{B}_2^\omega} (\mathbb{B}_2^\omega \multimap \mathbb{B}_2^*) !\mathbf{M}$.

Even if we cannot define a stream of the standard inductive natural numbers, we can have a stream of extended natural numbers. Let us define the latter first. Consider the functor $F(X) = \mathbf{1} \oplus X$. By Lemma 4.3.3, we have that F right-distributes over \S , and so by Theorem 4.3.3 we have that $(\overline{\mathbb{N}}, \text{out}_{\overline{\mathbb{N}}})$ is a weakly final F -coalgebra under \S , where $\overline{\mathbb{N}}$ denotes $\nu X. F(X)$. The inhabitants of the type $\overline{\mathbb{N}}$ correspond to the natural numbers extended with a limit element ∞ . We can think about $\text{out}_{\overline{\mathbb{N}}}$ as a predecessor function mapping 0 to $()$, n to $n - 1$ and ∞ to ∞ . We can define the addition of two extended natural numbers by considering the term:

$$\begin{aligned} \mathbf{M} &= \lambda x : \overline{\mathbb{N}} \otimes \overline{\mathbb{N}}. \text{let } \langle x_1, x_2 \rangle = x \text{ in } \left(\text{case } (\text{out}_{\overline{\mathbb{N}}} x_1) \text{ of} \right. \\ &\text{inj}_0(z) \rightarrow \text{case } (\text{out}_{\overline{\mathbb{N}}} x_2) \text{ of } \{ \text{inj}_0(z') \rightarrow \text{inj}_0(()) \mid \text{inj}_1(z') \rightarrow \text{inj}_1(\langle z, z' \rangle) \} \\ &\left. \mid \text{inj}_1(z) \rightarrow \text{inj}_1(\langle z, x_2 \rangle) \right). \end{aligned}$$

The term \mathbf{M} has type $\overline{\mathbb{N}} \otimes \overline{\mathbb{N}} \multimap \mathbf{1} \otimes (\overline{\mathbb{N}} \otimes \overline{\mathbb{N}})$. So, by Theorem 4.3.3 $\text{add} = \text{unfold}_{\overline{\mathbb{N}}} (\overline{\mathbb{N}} \otimes \overline{\mathbb{N}}) !\mathbf{M}$. For the extended natural numbers, we have a term $\text{coer}_{\overline{\mathbb{N}}} : \S \overline{\mathbb{N}} \multimap \overline{\mathbb{N}}$, this is given by Theorem 4.3.3 as $\text{coer}_{\overline{\mathbb{N}}} = \text{unfold}_{\overline{\mathbb{N}}} \overline{\mathbb{N}} !\text{out}_{\overline{\mathbb{N}}}$.

4.4 Alternative results on streams and real numbers

In this section, we review most of the alternative results of ICC either based on stream programming languages or characterizing complexity classes over real numbers.

4.4.1 Streams, parsimonious types and non-uniform complexity classes

In [Maz14], Mazza considers an infinitary affine lambda calculus as the completion of a finitary affine lambda calculus. This language is coupled with a *parsimonious* type system to characterize the class of non-uniform polynomial time computable decision problems P/poly . P/poly is the class of decision problems that can be computed by a family of non-uniform Boolean circuits of polynomial size; a family of circuits (C_n) being uniform if there is a polynomial time algorithm computing C_n for every input n . The intuition behind parsimony is to restrict the infinitary calculus to a subset of terms with a polynomial “*modulus of continuity*”. This line of work takes inspirations from the works about infinitary lambda calculus [Maz12, Maz14] and differential lambda calculus [ER03, ER08].

We present here the characterization introduced by Mazza and Terui in [MT15] that characterizes P/poly and also L/poly under some restrictions by combining a stream language and a parsimonious type system. An alternative syntax and alternative semantics with indexes simplifying the management of parsimony properties can also be found in [Maz15].

The parsimonious calculus

The essential features of the considered stream programming language are the following. Variables are separated into two different kinds, affine and exponential variables. Exponential variables x, y, z, \dots correspond to streams. A *box* construct allows us to define infinite streams in a finite syntax: $!_f(u_0, u_1, \dots, u_{k-1})$ which represents the stream $u_{f(0)} :: u_{f(1)} :: \dots$ for a given function $f : \mathbb{N} \rightarrow \mathbb{N}_k$, with $\mathbb{N}_k = \{0, \dots, k-1\}$. Terms of the language are defined by the following grammar:

$$t, u ::= \perp \mid a \mid x \mid \lambda a. t \mid t \ u \mid t \otimes u \mid !_f \bar{u} \mid t :: u \mid t[p := u],$$

where $f \in \mathbb{N} \rightarrow \mathbb{N}_k$, $k \geq 1$, and where \bar{u} denotes a list of k terms u_0, \dots, u_{k-1} .

Let variables $\mathbf{u}, \mathbf{v}, \dots$ range over *boxes* of the shape $!_f \bar{u}$ that are basically stream generators. The language includes a stream constructor $::$ as well as a pair constructor \otimes and the corresponding destructors/binders.

We adopt a convention similar to [ADL14] by using explicit substitutions $t[p := u]$, in which a pattern p is either a pair $a \otimes b$ or a stream $a_0 :: a_1 :: \dots :: a_{n-1} :: x$. In this latter case, we use sometimes the notation $p(x)$ to explicitly mention the exponential variable in the stream. This makes the syntax and semantics lighter by replacing the more verbose *let in* destructor.

In what follows, let $!u$ be a shorthand for the box of one element $!_f u_0$ with $f : \mathbb{N} \rightarrow \mathbb{N}_0$ defined by $\forall n \in \mathbb{N}, f(n) = 0$ and with $u_0 = u$. A term that does not contain non-uniform boxes is called uniform.

Example 4.4.1. *The head and tail functions on streams can be encoded by $\text{head} = \lambda a. b[b :: x := a]$ and $\text{tail} = \lambda a. c[c \otimes d := !x \otimes \perp][b :: x := a]$. Notice that the use of the pair constructor and destructor in the tail program is just a syntactical trick allowing us to weaken the affine variable b . One would have expected a tail program of the shape $\lambda a. !x[b :: x := a]$ but this would require the introduction of an extra weakening rule on affine variables.*

Definition 4.4.1 (Slice). *A slice of a term is obtained by removing all components but one from each box. Let $\mathcal{S}(t)$ be the set of slices of term t defined as follows.*

$$\begin{aligned}
 \mathcal{S}(\alpha) &= \{\alpha\} && \text{if } \alpha \in \{a, x\} \\
 \mathcal{S}(!_f(u_0, \dots, u_{k-1})) &= \bigcup_{i=0}^{k-1} \{!_f \nu_i; \nu_i \in \mathcal{S}(u_i)\} \\
 \mathcal{S}(t_1 \otimes t_2) &= \{\tau_1 \otimes \tau_2 \mid \tau_1 \in \mathcal{S}(t_1), \tau_2 \in \mathcal{S}(t_2)\} \\
 \mathcal{S}(t_1 \ t_2) &= \{\tau_1 \ \tau_2 \mid \tau_1 \in \mathcal{S}(t_1), \tau_2 \in \mathcal{S}(t_2)\} \\
 \mathcal{S}(t_1 :: t_2) &= \{\tau_1 :: \tau_2 \mid \tau_1 \in \mathcal{S}(t_1), \tau_2 \in \mathcal{S}(t_2)\} \\
 \mathcal{S}(t_1[p := t_2]) &= \{\tau_1[p := \tau_2] \mid \tau_1 \in \mathcal{S}(t_1), \tau_2 \in \mathcal{S}(t_2)\} \\
 \mathcal{S}(\lambda a.t) &= \{\lambda a.\tau_1 \mid \tau_1 \in \mathcal{S}(t)\}
 \end{aligned}$$

Example 4.4.2. $\mathcal{S}(!_f(x \otimes \lambda a.a, \lambda a.a \otimes x)) = \{!_f(x \otimes \lambda a.a), !_f(\lambda a.a \otimes x)\}$. As illustrated by this example, in general, a slice does not belong to the set of (syntactically correct) terms.

Definition 4.4.2 (Parsimonious term). *A term t is parsimonious if:*

1. all its slices are affine, i.e. each variable occurs at most once in a slice,
2. box subterms do not contain free affine variables,
3. all exponential variables belong to a box subterm.

Example 4.4.3. *To illustrate those points, let us see some (counter)-examples.*

- $\lambda a.a \otimes a$ is not parsimonious because of point 1. However, $\lambda a.(\lambda a.a)a$ is parsimonious.
- $!_f(x \otimes \lambda a.a, \lambda a.a \otimes x)$ is parsimonious. Indeed the affine variable a is not free and x occurs exactly once in each slice.
- $!_f(a, x, y, c)$ is not parsimonious because of point 2.
- $x \otimes !_f(y, z)$ is not parsimonious because of point 3.

Given a function $f : \mathbb{N} \rightarrow \mathbb{N}_k$, let f^{+i} , $i \in \mathbb{N}$, be the function in $\mathbb{N} \rightarrow \mathbb{N}_k$ defined by $\forall n \in \mathbb{N}, f^{+i}(n) = f(n+i)$. Given a box $\mathbf{u} = !_f(u_0, \dots, u_{k-1})$ and $k \in \mathbb{N}$, let \mathbf{u}^{+k} be the term equal to $!_{f+k}(u_0, \dots, u_{k-1})$.

Semantics

The one-step reduction corresponding to terms of the language is defined in Figure 4.7 relatively to a finite set sigma σ of stream patterns, i.e. $\sigma = \{a_1 :: x_1, \dots, a_n :: x_n\}$. Let $\mathcal{V}(\sigma)$ be equal to $\{a_1, x_1, \dots, a_n, x_n\}$.

As described in [MT15], the substitution $\{\{\mathbf{u}/x\}\}$ is non standard on boxes: if x occurs in a box $\mathbf{w} = !_g(w_0, \dots, w_{l-1})$ and $\mathbf{u} = !_f(u_0, \dots, u_{k-1})$ then $\mathbf{w}\{\{\mathbf{u}/x\}\} = !_h(v_0, \dots, v_{l-1})$ with $v_{ik+j} = w_i\{u_j/x\}$ and $h(n) = g(n)k + f(n)$. Moreover, in the rule (dup) and (aux), it is mandatory that $\{x_1, \dots, x_n\}$ are precisely the free exponential variables of $u_{f(0)}$. Finally, in the (aux) rule, the notation $t[\mathbf{u}]$ means that the box \mathbf{u} is a subterm of t . If not, then the rule becomes $t[x := v :: w] \xrightarrow{\{b_1 :: x_1, \dots, b_n :: x_n\}} t$.

As usual, a *context* is a term with at most one occurrence of a special symbol $\langle \cdot \rangle$, called *hole*. We denote by $C\langle t \rangle$ the result of substituting the term t to the hole in C , an operation which may capture variables. The one-step reduction rules are extended contextually: if $t \xrightarrow{\sigma} u$ and C does not bind variables of $\mathcal{V}(\sigma)$ then $C\langle t \rangle \xrightarrow{\sigma} C\langle u \rangle$. If $t \xrightarrow{\sigma \cup \{b :: x\}} u$ then $t[p(x) := v] \xrightarrow{\sigma} u[p(b :: x) := v]$.

The system reduction is defined to be the reflexive and transitive closure of $\xrightarrow{\emptyset}$.

$(\lambda a.t) u \xrightarrow{\emptyset} t\{u/a\}$	(beta)
$t[x := \mathbf{u}] \xrightarrow{\emptyset} t\{\{\mathbf{u}/x\}\}$	(merge)
$t[p := v] u \xrightarrow{\emptyset} (t u)[p := v]$	(com ₁)
$t[p := u[q := v]] \xrightarrow{\emptyset} (t[p := u])[q := v]$	(com ₂)
$t[a :: p := u :: v] \xrightarrow{\emptyset} t\{u/a\}[p := v]$	(cons)
$t[a \otimes b := u \otimes v] \xrightarrow{\emptyset} t\{u/a, v/b\}$	(pair)
$t[a :: p := \mathbf{u}] \xrightarrow{\{b_1::x_1, \dots, b_n::x_n\}} t\{u_{f(0)}\{b_1/x_1, \dots, b_n/x_n\}/a\}[p := \mathbf{u}^{+1}]$	(dup)
$t[\mathbf{u}][x := v :: w] \xrightarrow{\{b_1::x_1, \dots, b_n::x_n\}} t\{u_{f(0)}\{b_1/x_1, \dots, b_n/x_n, v/x\} :: \mathbf{u}^{+1}\}[x := w]$	(aux)

Figure 4.7: One-step reduction of the parsimonious calculus

Example 4.4.4. Consider the terms *head* and *tail* of Example 4.4.1 and let $!_f(\underline{0}, \underline{1})$ be a stream of Boolean numbers for some function $f : \mathbb{N} \rightarrow \{0, 1\}$ such that $f(2k) = 0$ and $f(2k + 1) = 1$, for each $k \in \mathbb{N}$, i.e. $!_f(\underline{0}, \underline{1}) = \underline{0} :: \underline{1} :: \underline{0} :: \underline{1} :: \dots$

As $\text{head} = \lambda a.b[b :: x := a]$, we have the following reduction.

$$\begin{aligned} \text{head } !_f(\underline{0}, \underline{1}) &\xrightarrow{\emptyset} b[b :: x := !_f(\underline{0}, \underline{1})] && \text{(beta)} \\ &\xrightarrow{\emptyset} \underline{0}[x := !_f(\underline{0}, \underline{1})] && \text{(dup)} \end{aligned}$$

As $\text{tail} = \lambda a.c[c \otimes d := !x \otimes \perp][b :: x := a]$, we have the following reduction.

$$\begin{aligned} \text{tail } !_f(\underline{0}, \underline{1}) &\xrightarrow{\emptyset} c[c \otimes d := !x \otimes \perp][b :: x := !_f(\underline{0}, \underline{1})] && \text{(beta)} \\ &\xrightarrow{\emptyset} !x[b :: x := !_f(\underline{0}, \underline{1})] && \text{(pair)} \\ &\xrightarrow{\emptyset} !x[x := !_f(\underline{0}, \underline{1})] && \text{(dup)} \\ &\xrightarrow{\emptyset} !x\{\{!_{f+1}(\underline{0}, \underline{1})/x\}\} = !_{f+1}(\underline{0}, \underline{1}) && \text{(merge)} \end{aligned}$$

The last equality is obtained by definition of non standard substitution. Indeed, in this context, suppose that $!_h(v_0, \dots, v_{l_{k-1}}) = !x\{\{!_{f+1}(\underline{0}, \underline{1})/x\}\}$. We have $h(n) = g(m)k + f^{+1}(n)$, with $g(m) = 0$, g being the function attached with the uniform box $!x$, and, consequently, $h(n) = f^{+1}(n)$ and $v_{i_{k+j}} = w_i\{u_j/x\} = x\{u_j/x\}$ with $u_0 = \underline{0}$ and $u_1 = \underline{1}$.

Parsimonious types

Types are defined inductively by:

$$A, B ::= \alpha \mid A \multimap B \mid A \otimes B \mid !A \mid \forall \alpha. A,$$

α being a type variable. As expected, the type $!A$ is for stream data of type A . The type system, $\text{nuPL}_{\forall \ell}$, is adapted from the uniform type system is defined in Figure 4.8. Typing judgments are of the form $\Gamma; \Delta \vdash t : A$ with Δ a typing environment for affine variables and Γ a typing

$$\begin{array}{c}
 \frac{}{\Gamma; \Delta, a : A \vdash a : A} \text{ (ax)} \quad \frac{}{\Gamma; \Delta \vdash \perp : A} \text{ (coveak)} \\
 \\
 \frac{\Gamma; \Delta, a : A \vdash t : B}{\Gamma; \Delta \vdash \lambda a.t : A \multimap B} \text{ (}\multimap\text{I)} \quad \frac{\Gamma; \Delta \vdash t : A \multimap B \quad \Gamma'; \Delta' \vdash u : A}{\Gamma, \Gamma'; \Delta, \Delta' \vdash t u : A} \text{ (}\multimap\text{E)} \\
 \\
 \frac{\Gamma; \Delta \vdash t : A \quad \Gamma'; \Delta' \vdash u : B}{\Gamma, \Gamma'; \Delta, \Delta' \vdash t \otimes u : A \otimes B} \text{ (}\otimes\text{I)} \quad \frac{\Gamma; \Delta \vdash u : A \otimes B \quad \Gamma'; \Delta', a : A, b : B \vdash t : C}{\Gamma, \Gamma'; \Delta, \Delta' \vdash t[a \otimes b := u] : C} \text{ (}\otimes\text{E)} \\
 \\
 \frac{\Gamma, p : A; \Delta, a : A \vdash t : B}{\Gamma, a :: p : A; \Delta \vdash t : B} \text{ (abs)} \quad \frac{\Gamma; \Delta \vdash t : A \quad \Gamma'; \Delta' \vdash u : !A}{\Gamma, \Gamma'; \Delta, \Delta' \vdash t :: u : !A} \text{ (coabs)} \\
 \\
 \frac{; \bar{a} : \bar{A} \vdash \bar{u}_0 : A \quad \dots \quad ; \bar{a} : \bar{A} \vdash \bar{u}_{k-1} : A}{\bar{x} : \bar{A}; \vdash !_f \bar{u} \{ \bar{x} / \bar{a} \} : !A} \text{ (!I)} \quad \frac{\Gamma; \Delta \vdash u : !A \quad \Gamma', p : A; \Delta' \vdash t : B}{\Gamma, \Gamma'; \Delta, \Delta' \vdash t[p := u] : B} \text{ (!E)} \\
 \\
 \frac{\Gamma; \Delta \vdash t : A \quad \alpha \notin FV(\Gamma \cup \Delta)}{\Gamma; \Delta \vdash t : \forall \alpha. A} \text{ (}\forall\text{I)} \quad \frac{\Gamma; \Delta \vdash t : \forall \alpha. A}{\Gamma; \Delta \vdash t : A[B/\alpha] \quad B \text{ is !-free}} \text{ (}\forall\text{E)}
 \end{array}$$

Figure 4.8: Non-uniform parsimonious logic type system

environment for patterns $p(x)$; the variables in Γ and Δ being distinct. In the (!I) rule of Figure 4.8, the variables \bar{x} are fresh. It implies that they do not appear free in each u_i . The type system, **nuPL** is the restriction of **nuPL**_{∇ℓ} where rules (∇I) and (∇E) are withdrawn.

Example 4.4.5. *The tail program of Example 4.4.1 can be typed in **nuPL**_{∇ℓ} as follows.*

$$\begin{array}{c}
 \frac{}{; e : A \vdash e : A} \text{ (ax)} \\
 \frac{}{x : A; \vdash !x : !A} \text{ (!I)} \quad \frac{}{; \vdash \perp : A} \text{ (coveak)} \\
 \frac{}{x : A; \vdash !x \otimes \perp : !A \otimes A} \text{ (!I)} \quad \frac{}{; c : !A, d : A, b : A \vdash c : !A} \text{ (ax)} \\
 \frac{}{; a : !A \vdash a : !A} \text{ (ax)} \quad \frac{x : A; b : A \vdash c[c \otimes d := !x \otimes \perp] : !A}{b :: x : A; \vdash c[c \otimes d := !x \otimes \perp] : !A} \text{ (abs)} \\
 \frac{}{; \vdash \lambda a.c[c \otimes d := !x \otimes \perp][b :: x := a] : !A \multimap !A} \text{ (}\multimap\text{I)} \quad \frac{}{; \vdash \lambda a.c[c \otimes d := !x \otimes \perp][b :: x := a] : !A \multimap !A} \text{ (}\multimap\text{I)}
 \end{array}$$

Main results

The type system implies that each typable term is parsimonious. The converse obviously does not hold.

Proposition 4.4.1 ([MT15]). *Given a term t , if there are a typing environment Δ and a type A such that $;\Delta \vdash t : A$ then t is parsimonious.*

Hence typable terms have a behavior with a polynomial “modulus of continuity”. The restriction to **nuPL** avoid the programmer from being able to compute a simple iteration scheme within the language. Let $\llbracket \mathbf{nuPL} \rrbracket$ and $\llbracket \mathbf{nuPL}_{\forall \ell} \rrbracket$ be the classes of language decidable by terms typable in **nuPL** and **nuPL**_{∀ℓ}, respectively.

Theorem 4.4.1 ([MT15]). $\llbracket \mathbf{nuPL} \rrbracket = \mathbf{P/poly}$ and $\llbracket \mathbf{nuPL}_{\forall \ell} \rrbracket = \mathbf{L/poly}$.

This type discipline was also adapted in [Maz15] to characterize **Logspace** by restricting the language to uniform boxes (mainly constant stream). The parsimonious methodology is very convenient to capture small complexity classes with programs computing over streams. One of its drawback is that its computations on stream behave as a transducer. Hence each stream computation only depends on a finite portion of the input streams, which makes the calculus not straightforwardly adaptable to characterize polynomial time over the reals or higher-order complexity classes such as **BFF**₂.

4.4.2 Function algebra characterizations of polynomial time over the reals

An interesting function algebra based characterization of the functions computable in polynomial time over real numbers based on Bellantoni and Cook function algebra (see Theorem 1.2.2) has been provided by Bournez, Hainry, and Gomaa [BGH11]. This characterization does not focus precisely on the class $\mathbf{FP}(\mathbb{R})$ introduced in Definition 4.2.10 but on a strict subset $\mathbf{FP}(\mathbb{R}) \cap ([0, 1] \rightarrow \mathbb{R}^+)$, the class of functions from $[0, 1]$ to \mathbb{R}^+ that are computable by a Machine in polynomial time.

Definition 4.4.3. Let \mathcal{W} be the least class of functions containing the constant functions 0, 1, the binary addition $+(; x, y) = x + y$ and binary subtraction $-(; x, y) = x - y$, the projection functions $\pi_j^n(; x_1, \dots, x_n) = x_j$, the conditional function c defined by $c(; x, y, z) = xy + (1 - x)z$, the continuous parity function $par(x;) = \max(0, 2/(\pi \sin(\pi x)))$, and the continuous predecessor function $p(x;) = \int_0^{x-1} par(t;)dt$, and closed under safe composition (**SCOMP**) and Safe Integration (**SI**):

$$\begin{aligned} \mathbf{SCOMP}(f, \bar{g}, \bar{h})(\bar{x}; \bar{y}) &= f(\bar{g}(\bar{x};); \bar{h}(\bar{x}; \bar{y})), \\ \mathbf{SI}(f, h_0, h_1)(0, \bar{y}; \bar{z}) &= f(\bar{y}; \bar{z}), \\ \partial_x \mathbf{SI}(f, h_0, h_1)(x, \bar{y}; \bar{z}) &= par(x;) [h_1(p(x;), \bar{y}; \bar{z}, \mathbf{SI}(f, h_0, h_1)(p(x;), \bar{y}; \bar{z})) - \mathbf{SI}(f, h_0, h_1)(2p(x;), \bar{y}; \bar{z})] \\ &\quad + par(x - 1;) [h_0(p'(x;), \bar{y}; \bar{z}, \mathbf{SI}(f, h_0, h_1)(p'(x;), \bar{y}; \bar{z})) - \mathbf{SI}(f, h_0, h_1)(2p'(x;), \bar{y}; \bar{z})], \end{aligned}$$

with $p'(x;) = p(x - 1;) + 1$. In other words, $\mathcal{W} = [0, 1, +, -, \pi_j^n, c, par, p; \mathbf{SCOMP}, \mathbf{SI}]$.

\mathcal{W} is a direct extension of the BC algebra to the real numbers. Indeed, the consumption of one bit i in the BC algebra is replaced by one application of the continuous predecessor function p and the choice of the contextual function h_i is simulated using the parity function par . As a consequence, the basic functions are total functions over \mathbb{R}^+ and hence any function of the algebra is a total function over \mathbb{R}^+ . Moreover, the class \mathcal{W} preserves integers (and natural numbers) and, consequently, its restriction to natural numbers provides a characterization of **FP**.

Theorem 4.4.2 ([BGH11]). $\mathcal{W} \cap (\mathbb{N} \rightarrow \mathbb{N}) = \mathbf{FP}$.

It is (strictly) included in the class of polynomial time computable functions over the real numbers.

Theorem 4.4.3 ([BGH11]). $\mathcal{W} \subsetneq \mathbf{FP}(\mathbb{R})$.

The characterization of $\text{FP}(\mathbb{R}) \cap ([0, 1] \rightarrow \mathbb{R}^+)$ is obtained as follows.

Theorem 4.4.4 ([BGH11]). $\text{FP}(\mathbb{R}) \cap ([0, 1] \rightarrow \mathbb{R}^+)$ is exactly the the class of functions that either are Lipschitz and \mathcal{W} -definable or that are n^k -smooth and n^k - \mathcal{W} -definable.

Remember that a function f is Lipschitz if there exists a constant $k \geq 0$ such that for all x, y , $|f(x) - f(y)| \leq k|x - y|$. In the above theorem, we have made an arbitrary choice not to describe the notions of definability and smoothness. Definability is related to function approximation and n^k -smoothness is related to a polynomial modulus of continuity. The full details can be found in [BGH11].

It is worth noticing that related works [BH04, CO07] provide also interesting function algebra characterizations of recursive functions (*i.e.* computable functions) over real numbers.

4.4.3 The BSS model

The papers [BCJDNM05, BCJDNM06] provide function algebra characterizations of complexity classes in the Blum-Shub-Smale (BSS) model, which is another model for describing computations over real numbers and their complexity.

The BSS model [BSS88] consists in RAMs, whose registers can store arbitrary real numbers, that compute constant time arithmetic operations on real numbers following a finite list of instructions. Such machines take inputs from $\mathbb{R}^\infty = \cup_{n=1}^\infty \mathbb{R}^n$ and halt by returning an output in \mathbb{R}^∞ or loop forever. In what follows, let a be an element of \mathbb{R} , let \bar{x} be an element of \mathbb{R}^∞ and let $a.\bar{x}$ be the element of \mathbb{R}^∞ whose first component is a followed by \bar{x} . Let ϵ be the empty tuple.

Let M be a BSS machine and $\llbracket M \rrbracket$ be the function associating each input in \mathbb{R}^∞ to the corresponding output in \mathbb{R}^∞ . A function $f : (\mathbb{R}^\infty)^k \rightarrow \mathbb{R}^\infty$ is computable in the BSS model if there exists a machine M such that $f = \llbracket M \rrbracket$.

A BSS machine runs in polynomial time if there is a polynomial $P \in \mathbb{N}[X]$ such that for any input $\bar{x} \in \mathbb{R}^\infty$ the machine produces an output in $P(|\bar{x}|)$ steps.²⁹ Let $\text{FP}_{\mathbb{R}}$ be the class of functions computable by BSS machines in polynomial time.

Theorem 4.4.5 ([BCJDNM05]³⁰). *The least class of functions containing the constant functions 0 and 1, the head, tail and cons functions over sequences of real numbers defined by $\text{head}(\epsilon; a.\bar{x}) = a$, $\text{tail}(\epsilon; a.\bar{x}) = \bar{x}$, $\text{head}(\epsilon; \epsilon) = \text{tail}(\epsilon; \epsilon) = \epsilon$, $\text{cons}(\epsilon; a.\bar{x}, \bar{y}) = a.\bar{y}$, and $\text{cons}(\epsilon; \epsilon, \bar{y}) = \bar{y}$, the projection functions $\pi_i^n(\bar{x}_1, \dots, \bar{x}_n) = \bar{x}_i$, $i \in [1, n]$, the arithmetic operations \otimes , \oplus and \leq defined by $\oplus(\epsilon; a.\bar{x}, b.\bar{y}) = (a + b).\bar{y}$, $\otimes(\epsilon; a.\bar{x}, b.\bar{y}) = (a \times b).\bar{y}$, $\leq(\epsilon; a.\bar{x}, b.\bar{y}) = i.\bar{y}$, with $i = 1$ if $a \leq b$, and $i = 0$ otherwise, the conditional function $C(\epsilon; a.\bar{x}, \bar{y}, \bar{z}) = \bar{y}$, if $a = 1$, $C(\epsilon; a.\bar{x}, \bar{y}, \bar{z}) = \bar{z}$ otherwise, and closed under safe composition (SCOMP) and Safe Recursion (SR):*

$$\begin{aligned} \text{SCOMP}(f, \bar{g}, \bar{h})(\bar{x}; \bar{y}) &= f(\bar{g}(\bar{x}); \bar{h}(\bar{x}; \bar{y})), \\ \text{SR}(f, g)(\epsilon, \bar{y}; \bar{z}) &= g(\bar{y}; \bar{z}), \\ \text{SR}(f, g)(a.\bar{x}, \bar{y}; \bar{z}) &= f(\bar{x}, \bar{y}; \bar{z}, \text{SR}(f, g)(\bar{x}, \bar{y}; \bar{z})), \end{aligned}$$

is exactly $\text{FP}_{\mathbb{R}}$. In other words, $[0, 1, \text{head}, \text{tail}, \text{cons}, \pi_i^n, \otimes, \oplus, \leq, C; \text{SCOMP}, \text{SR}] = \text{FP}_{\mathbb{R}}$.

²⁹The size of \bar{x} is the dimension of the tuple.

³⁰Here we consider \mathbb{R} with a ring structure. Hence subtraction and division are not included. They could have been considered as the result of [BCJDNM05] holds for arbitrary structures. In this case, for Theorem 4.4.5 to hold, subtraction and division have to be added as basic operations in the corresponding function algebra. $+$ and \times denote respectively the standard addition and standard multiplication operations on real numbers.

Characterizations of other interesting complexity classes based on the BSS model such as parallel polynomial time and classes of the polynomial hierarchy can be found in [BCJDNM05] and [BCJDNM06], respectively. For the interested reader, a characterization of $\text{FP}_{\mathbb{R}}$ based on LAL was also studied in [BP06].

Chapter 5

Research perspectives

This chapter presents the research directions in ICC that I would like to explore in the future or that, in my opinion, correspond to the main research issues that should be addressed in the next decades. In Section 5.1 and Section 5.2, I mention some open issues and research directions in the context of probabilistic programs and quantum computing, respectively. In Section 5.3, I discuss the open issues related to languages for real numbers computations and with complexity certificates. In Section 5.4, I mention the issue of finding a decidable theory for type-2 polynomial time complexity. Finally, in Section 5.5, I discuss two typing disciplines of interest, sized types and intersection types, that are used to characterize termination of some lambda-calculi and mention some related open issues of interest from a complexity viewpoint.

5.1 Probabilistic models

The notion of probabilistic programming language (and model) has had a renewed popularity due to the need for such applications in algorithmics, robotics, and cryptography. Current probabilistic languages consist in higher-order functional languages with sampling and conditioning instructions.³¹

Like classical programs, probabilistic programs are in need of tools and techniques for formalizing and reasoning on their semantics. Several semantics aspect of probabilistic programs have already been studied (denotational semantics [Koz79], operational semantics [BDLGS16], uniqueness, normalization, confluence, and standardization of the lambda calculus with choice [Fag19, FRDR19], ...). They also require some verification techniques to be developed to certify properties such as termination. Termination with probability 1 is called, *almost sure termination* and if, in addition, the mean length of a derivation is finite then it is called *positive almost sure termination*. This latter notion has been deeply studied for TRSs with probabilistic rewrite rules by [BG05, BG06, Gna07, ADLY18] and almost sure termination for probabilistic higher-order languages has been studied in [KDLG19, DLG19] .

There are only a few works for verification techniques based on ICC methods. [DLT15] provides a first characterization of PP, the class of decision problems solvable by a probabilistic TM in polynomial time, with an error probability of less than 1/2 for all instances. A less implicit extension to BPP, the class of decision problems solvable by a probabilistic TM in polynomial time with an error probability less than 1/3 for all instances, is also considered. Although pioneering,

³¹Conditioning consists in allowing to add information about observed events into the program that may influence the posterior probability distribution.

this work is not satisfactory insofar as the PP class is not really interesting in this context (as error is too close from success) and the characterization of BPP is not sufficiently implicit.

More recently, Avanzini, Moser, and Schaper have developed an average case runtime analysis for imperative languages [AMS19]. It is inspired by the ert-calculus [KKMO16], a method allowing to infer expected runtimes of probabilistic programs. The paper [ADLG19] also provides a sound and complete average case runtime analysis of higher-order functional programs with respect to the average case polytime TMs. Intuitively, the restriction of this class to decision problems is strictly included in ZPP, the class of zero-error polynomial time probabilistic programs, as the non-polynomial executions may be transformed into errors and, consequently, this work is an interesting improvement on existing methods.

As the characterization of [ADLG19] relies on an affine type system, it would be of interest to study extensions of the tiering method in an imperative and probabilistic setting in order to characterize this complexity class and to study to which extent a pure ICC characterization of ZPP can be obtained.

5.2 Quantum computations

Quantum computing is a computational paradigm which takes advantage of quantum mechanics to perform computations. An important number of quantum algorithms (Shor's [Sho97], Grover's [Gro96],...) have been shown to be more efficient than their classical counterpart in terms of time complexity. Several interesting works on the semantics properties of quantum programs have been carried out in recent years [Sel04]. However, while there are plenty of tools to analyze automatically the complexity of classical programs, only a few works have been carried out for quantum programs.

The work by Dal Lago et al. [DLMZ10], presented in Subsection 3.2.3, was to our knowledge the first application of ICC techniques to the quantum paradigm. However, it is not fully satisfactory for two main reasons. First, as already mentioned in Subsection 3.2.3, measurements are not allowed as a basic construct within the programming language. Consequently, this work cannot be straightforwardly adapted to mainstream quantum programming languages such as QPL (Quantum Programming Language) of [Sel04]. Second, most semantics restrictions on the definition of the captured quantum complexity classes are also holding on the notion of computed function or accepted language (e.g. Definition 3.2.2) and, hence, this weakens the purity of the characterization. One solution might be to consider the (unpublished) work of [Yam18] which provides the first function algebra characterization of the quantum complexity class (F)BQP. In this work, the semantics requirements are less external in the sense that a function is in FBQP if and only if it can be “*approximated*” polynomially by some function definable in the function algebra. Here the weakness is the extra requirement of this external polynomial that one could get rid off using standard safe-like function algebra. Another study of interest would then be to see whether the tiering approach can be adapted to a fragment of QPL to characterize similar complexity classes.

Last but not least, a well-suited complexity analysis of quantum programs should take into account the entanglement of qubits during program execution. Indeed, most of the algorithms improving the complexity of their classical counterpart (e.g. [Sho97, Gro96]) make use of entanglement to improve the algorithmic speed. Consequently, entanglement seems to be correlated to this speed up and an adequate study of quantum programs complexity should take a measure of entanglement into account. One possible direction is to consider the interesting work of Perdrix which has developed an abstract interpretations based static analysis for getting an

approximation of entangled qubits [Per08].

5.3 Feasible computations over the reals

Some characterizations of the class of real functions computable in polynomial time $\text{FP}(\mathbb{R})$ have been provided in Section 4.2 and Section 4.4 ([FHHP10, BGH11, FHHP15]). They have also shown to be equivalent to other well-know computational models over the reals such as the General Purpose Analog Computer (GPAC)[BGP16]. However, these characterizations suffer from several drawbacks: either they provide a non-natural way to write programs (*i.e.* function algebra with non standard operators and recursion schemata) or they involve criteria that are not tractable (type-2 higher-order polynomials over stream programs).

Characterizing the complexity class $\text{FP}(\mathbb{R})$ is still challenging as, by soundness, programs computing functions of $\text{FP}(\mathbb{R})$ correspond to programs that can give an approximation of the output at precision n in polynomial time in n , which is the intuitive and rational way to understand the complexity of a function over real numbers. For being effective, works characterizing the class $\text{FP}(\mathbb{R})$ should not constrain the programmer in their way of writing down programs but should rather provide a certificate (type) that the program has the good complexity properties. Hence one issue of interest is to consider a standard functional programming language on streams (*e.g.* streams of signed digits) encoding real numbers and to develop a type system sound and complete for $\text{FP}(\mathbb{R})$. One suggestion towards the completion of this work would be to consider the DLAL system of Baillot and Terui introduced in Section 2.1 and to combine it with stream data with good properties such as productivity [Sev17, AM13a, CBGB15].

5.4 A decidable theory for type-2 polynomial time

As mentioned in Subsection 4.2.1, several characterizations of higher complexity, more precisely, type-2 polynomial time complexity were studied in the last decades. All these characterizations suffer from three main problems. They rely on some external or explicit bounds [Con73, Meh76, CK90, IRK01, KS18, KS19]. Consequently, programming in such a paradigm is very hard as program behaviors are unpredictable (in particular the oracles answers cannot be predicted), or they are using machines or non natural function scheme and cannot be used in practice [Coo92, CU93], or they are in need of an undecidable check on constraints (*e.g.*, inequalities over higher-order polynomials) [KC91, KC96, HP17].

A main issue of interest for this complexity theory is hence to provide a tractable characterization based on realistic programming language. One direction is to consider the characterization of [KS18], showing that Oracle Polynomial Time (OPT) (see [Coo92]), together with a restriction on the number of *lookahead revisions* (the number of time the size of an oracle input can increase is bounded by a constant), characterize BFF_2 (under some lambda closure). Such a semantics property could be enforced by techniques such as tiering or light affine type systems and, hence, would allow us to characterize BFF_2 in a tractable manner on a realistic programming (imperative or functional) language.

5.5 Other typing disciplines

Several other typing techniques have been used for termination and complexity purposes.

One of the most successful techniques is *sized types* that have been used to prove correctness properties of reactive systems [HPS96], size based termination [CK01, BGR08], probabilistic termination [DLG19], and space upper bounds [Vas08].

The line of work on the use of dependent types [DLG11, DLP14] for inferring program complexity properties is related to this approach. Indeed, as mentioned by Dal Lago, “*linear dependent types can be seen as a way to inject precision and linearity into sized types*”.

Sized types have been adapted to functional languages to infer resource upper bounds in practice [ADL17]. This methodology can be fully automated and a relative completeness result is also stated but it merely focuses on soundness properties and on extending the expressive power (some examples of programs that were not captured are provided and no characterization of complexity class is studied). In [BG18], sized types have been combined with light linear logic to characterize the full Grzegorzczuk’s hierarchy, including FP. One question of interest is whether such a technique could also be combined to other ICC techniques such as tiering and interpretation (at least for first order). Another question of interest is also whether the work of [BG18] could be transferred to other complexity classes (polynomial space and subpolynomial classes).

Another technique of interest is *intersection types*. Intersection types have been used to show strong normalization of the lambda-calculus for the non-idempotent framework (i.e. provided that the intersection of a type with itself is not the identity) [DC05, BL11, BKV17, DC18]. Characterizations of head normalizing terms and strongly normalizing terms have also been adapted to non-idempotent intersection types and union types of the lambda-mu-calculus in [KV17]. In this setting upper bounds can be derived on the (head)-reduction length. Intersection types have also been applied to the setting of the probabilistic lambda-calculus [BDL18], where the probability of a term to terminate is characterized as the least upper bound of the “weight” of its typing derivations. Non-idempotent intersection types are used for characterizing terms normalizing in elementary time in [BRDR15]. For that purpose, the considered types are also non-associative. The non-associativity entails a stratification that is reminiscent of the stratification in light logics. In this framework, characterizations of the most popular complexity classes remain open issues.

Glossary

Alogtime : U_E^* uniform NC^1
BC : Bellantoni and Cook function algebra
BFF : Basic Feasible Functionals
BFF₂ : Basic Feasible Functionals at order 2
BLL : Bounded Linear Logic
BQP : Bounded error Quantum Polynomial time
BPP : The class of problems computable in polytime with bounded-error by a probabilistic TM
BSS : Blum-Shub-Smale model
BTLP : Bounded Type Loop Programs
CBN : Call-By-Name
CBV : Call-By-Value
CPS : Continuation-Passing Style
DLAL : Dual Light Affine Logic
DP : Dependency Pair
DPG : Dependency Pair Graph
DPI : Dependency Pair Interpretation
EAL : Elementary Affine Logic
ELL : Elementary Linear Logic
EQP : Exact Quantum Polynomial time
EXPTIME : The class of problems computable in exponential time by a deterministic TM
FBQP : Functional BQP
FP(\mathbb{R}) : The class of functions computable in polynomial time over the reals
FP $_{\mathbb{R}}$: The class of functions computable by BSS machines in polynomial time
FMT : Finite Model Theory
FP : The class of functions computable in polynomial time by a deterministic TM
FPSPACE : The class of functions computable in polynomial space by a deterministic TM
GPAC : General Purpose Analog Computer
HO : Higher-Order
HOP : Higher-Order Polynomial
I : Interpretation
ICC : Implicit Computational Complexity
L : The class of problems computable in logarithmic space by a deterministic TM
L/poly : The class of decision problems computable by branching programs of polynomial size
LAL : Light Affine Logic
LALC : Light Affine Lambda Calculus
LL : Linear Logic
LLL : Light Linear Logic
LLPO : Light Lexicographic Path Ordering

Logspace : L
LPO : Lexicographic Path Ordering
MLSTA : ML Soft Type Assignment
MPO : Multiset Path Ordering
 NC^k : Decision problems computable by U_E^* uniform circuits of polynomial size and \log^k depth
NC : $\cup_{k \geq 0} \text{NC}^k$
NP : The class of decision problems computable in polynomial time by a non deterministic TM
OTM : Oracle Turing Machine
OO : Object Oriented
OPT : Oracle Polynomial Time
P : The class of decision problems computable in polynomial time by a deterministic TM
P/poly : The class of decision problems computable by families of polynomial size circuits
PCF : The language of Programming Computable Functions
POP : Product Path Ordering
POP* : Polynomial Path Ordering
PP : The class of decision problems computable in polynomial time by a probabilistic TM
PSPACE : The class of problems computable in polynomial space by a deterministic TM
QBF : Quantified Boolean Formula
QI : Quasi-interpretation
QPL : Quantum Programming Language
QTM : Quantum Turing Machine
RAM : Random Access Machine
RPO : Recursive Path Ordering
SAL : Soft Affine Logic
SAT : Satisfiability problem
SCP : Size Change Principle
SI : Sup-interpretation
SLL : Soft Linear Logic
SMT : Satisfiability Modulo Theory
sPOP* : small Polynomial Path Ordering
STA : Soft Type Assignment
STTRS : Simply Typed TRS
TM : Turing Machine
TRS : Term Rewrite System / Term Rewriting System
 U_E^* : A condition of uniformity for families of Boolean circuits (see [Ruz81])
ZPP : Zero-error Probabilistic Polynomial time
ZQP : Zero-error Quantum Polynomial time

List of Figures

1.1	Typing rules for LAL (version from [BT09])	41
1.2	Decidability and complexity of the type inference	49
1.3	Decidability and complexity of the synthesis problem	50
2.1	Typing rules for DLAL	59
2.2	Typing rules for STA	62
2.3	Recursive path ordering	70
3.1	Big step operational semantics of imperative programs	91
3.2	Tier-based imperative type system	92
3.3	Small step operational semantics of multi-threads	95
3.4	Tier-based multi-threads typing rule	96
3.5	Small step operational semantics with deterministic scheduling	98
3.6	Small step operational semantics of environments	99
3.7	Tier-based processes type system	100
3.8	Admissible types for unary operators	102
3.9	Tier-based OO type system	108
3.10	LLL-based well-formedness rules for $\lambda^{l,\$,\parallel}$	116
3.11	Operational semantics of LHO π	118
3.12	SLL-based process type system	118
3.13	Standard reduction rules of SQ	120
3.14	SQ type system	120
3.15	Type system for the higher-order functional program	126
3.16	Higher-order interpretation of a term	128
3.17	Big step operational semantics of OO programs	131
4.1	First order lazy operational semantics	140
4.2	BTLP grammar	152
4.3	Syntax of LALC	154
4.4	Semantics of LALC	155
4.5	Type system for LALC	156
4.6	LALC extended with distributions	161
4.7	One-step reduction of the parsimonious calculus	165
4.8	Non-uniform parsimonious logic type system	166

Bibliography

- [AAB⁺13] Roberto M. Amadio, Nicolas Ayache, François Bobot, et al. Certified Complexity (CerCo). In *FOPARA*, pages 1–18. Springer, 2013.
- [AAG⁺07a] Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, and Damiano Zanardini. Cost analysis of Java bytecode. In *ESOP*, pages 157–172. Springer, 2007.
- [AAG⁺07b] Elvira Albert, Puri Arenas, Samir Genaim, German Puebla, and Damiano Zanardini. COSTA: design and implementation of a cost and termination analyzer for Java bytecode. In *FMCO*, pages 113–132. Springer, 2007.
- [AAG⁺12] Elvira Albert, Puri Arenas, Samir Genaim, German Puebla, and Damiano Zanardini. Cost analysis of object-oriented bytecode programs. *Theoretical Computer Science*, 413(1):142–159, 2012.
- [AARG12] Nicolas Ayache, Roberto M. Amadio, and Yann Régis-Gianas. Certifying and reasoning on cost annotations in C programs. In *FMICS*, pages 32–46. Springer, 2012.
- [AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.
- [ABT07] Vincent Atassi, Patrick Baillot, and Kazushige Terui. Verification of Ptime reducibility for system F terms: type inference in dual light affine logic. *Logical Methods in Computer Science*, 3(4), 2007.
- [ACGDZJ04] Roberto M. Amadio, Solange Coupet-Grimal, Silvano Dal Zilio, and Line Jakubiec. A functional scenario for bytecode verification of resource bounds. In *CSL*, pages 265–279. Springer, 2004.
- [ACJ⁺18] Elvira Albert, Jesús Correas, Einar Broch Johnsen, Ka I Pun, and Guillermo Román-Díez. Parallel cost analysis. *Transactions on Computational Logic*, 19(4):31, 2018.
- [AD06] Roberto M. Amadio and Frédéric Dabrowski. Feasible reactivity for synchronous cooperative threads. *Electronic Notes in Theoretical Computer Science*, 154(3):33–43, 2006.
- [ADL14] Beniamino Accattoli and Ugo Dal Lago. Beta reduction is invariant, indeed. In *CSL and LICS*, number 8, pages 1–10. ACM, 2014.
- [ADL17] Martin Avanzini and Ugo Dal Lago. Automating sized-type inference for complexity analysis. In *ICFP*, volume 1, pages 1–29. ACM, 2017.

- [ADL18] Martin Avanzini and Ugo Dal Lago. On sharing, memoization, and polynomial time. *Information and Computation*, 261:3–22, 2018.
- [ADLG19] Martin Avanzini, Ugo Dal Lago, and Alexis Ghyselen. Type-based complexity analysis of probabilistic functional programs. In *LICS*, pages 1–13. IEEE, 2019.
- [ADLM15] Martin Avanzini, Ugo Dal Lago, and Georg Moser. Analysing the complexity of functional programs: higher-order meets first-order. In *ICFP*, volume 50, pages 152–164. ACM, 2015.
- [ADLY18] Martin Avanzini, Ugo Dal Lago, and Akihisa Yamada. On probabilistic term rewriting. In *FLOPS*, pages 132–148. Springer, 2018.
- [ADZ04] Roberto M. Amadio and Silvano Dal Zilio. Resource control for synchronous cooperative threads. In *CONCUR*, pages 68–82. Springer, 2004.
- [AEM15] Martin Avanzini, Naohi Eguchi, and Georg Moser. A new order-theoretic characterisation of the polytime computable functions. *Theoretical Computer Science*, 585:3–24, 2015.
- [AFRD15] Elvira Albert, Jesús Correas Fernández, and Guillermo Román-Díez. Non-cumulative resource analysis. In *TACAS*, pages 85–100. Springer, 2015.
- [AG00] Thomas Arts and Jürgen Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236(1-2):133–178, 2000.
- [AGGZ07] Elvira Albert, Samir Genaim, and Miguel Gomez-Zamalloa. Heap space analysis for Java bytecode. In *ISMM*, pages 105–116. ACM, 2007.
- [AGGZ13] Elvira Albert, Samir Genaim, and Miguel Gómez-Zamalloa. Heap space analysis for garbage collected languages. *Science of Computer Programming*, 78(9):1427–1448, 2013.
- [AKM09] James Avery, Lars Kristiansen, and Jean-Yves Moyén. Static complexity analysis of higher order programs. In *FOPARA*, pages 84–99. Springer, 2009.
- [All91] Bill Allen. Arithmetizing uniform NC. *Annals of Pure and Applied Logic*, 53(1):1–50, 1991.
- [AM08] Martin Avanzini and Georg Moser. Complexity analysis by rewriting. In *FLOPS*, pages 130–146. Springer, 2008.
- [AM09] Martin Avanzini and Georg Moser. Dependency pairs and polynomial path orders. In *RTA*, pages 48–62. Springer, 2009.
- [AM10a] Martin Avanzini and Georg Moser. Closing the gap between runtime complexity and polytime computability. In *RTA*, volume 6 of *LIPICs*, pages 33–48. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010.
- [AM10b] Martin Avanzini and Georg Moser. Complexity analysis by graph rewriting. In *FLOPS*, pages 257–271. Springer, 2010.
- [AM13a] Robert Atkey and Conor McBride. Productive coprogramming with guarded recursion. In *ICFP*, volume 48, pages 197–208. ACM, 2013.

-
- [AM13b] Martin Avanzini and Georg Moser. Tyrolean complexity tool: Features and usage. In *RTA*, volume 21 of *LIPICs*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013.
- [AM16] Martin Avanzini and Georg Moser. A combination framework for complexity. *Information and Computation*, 248:22–55, 2016.
- [Ama03] Roberto M. Amadio. Max-plus quasi-interpretations. In *TLCA*, pages 31–45. Springer, 2003.
- [Ama05] Roberto M. Amadio. Synthesis of max-plus quasi-interpretations. *Fundamenta Informaticae*, 65(1-2):29–60, 2005.
- [AMS16] Martin Avanzini, Georg Moser, and Michael Schaper. TcT: Tyrolean complexity Tool. In *TACAS*, pages 407–423. Springer, 2016.
- [AMS19] Martin Avanzini, Georg Moser, and Michael Schaper. Modular runtime complexity analysis of probabilistic while programs. DICE-FOPARA workshops, 2019.
- [AR02] Andrea Asperti and Luca Roversi. Intuitionistic light affine logic. *Transactions on Computational Logic*, 3(1):137–175, 2002.
- [Asp98] Andrea Asperti. Light affine logic. In *LICS*, pages 300–308. IEEE, 1998.
- [Ava08] Martin Avanzini. POP* and semantic labeling using SAT. In *ESSLLI*, pages 155–166. Springer, 2008.
- [B⁺84] Hendrik P. Barendregt et al. *The Lambda Calculus*, volume 3. North-Holland Amsterdam, 1984.
- [BA02] Amir Ben-Amram. General size-change termination and lexicographic descent. In *The essence of computation*, pages 3–17. Springer, 2002.
- [BAG13] Amir Ben-Amram and Samir Genaim. On the linear ranking problem for integer linear-constraint loops. In *POPL*, volume 48, pages 51–62. ACM, 2013.
- [BAG17] Amir Ben-Amram and Samir Genaim. On multiphase-linear ranking functions. In *CAV*, pages 601–620. Springer, 2017.
- [BAGM12] Amir Ben-Amram, Samir Genaim, and Abu Naser Masud. On the termination of integer loops. *Transactions on Programming Languages and Systems*, 34(4):16, 2012.
- [Bai02] Patrick Baillot. Checking polynomial time complexity with types. In *Foundations of Information Technology in the Era of Network and Mobile Computing*, pages 370–382. Springer, 2002.
- [Bai04a] Patrick Baillot. Stratified coherence spaces: a denotational semantics for light linear logic. *Theoretical Computer Science*, 318(1-2):29–55, 2004.
- [Bai04b] Patrick Baillot. Type inference for light affine logic via constraints on words. *Theoretical Computer Science*, 328(3):289–323, 2004.

- [Bai08] Patrick Baillot. Logique linéaire, types et complexité implicite. Mémoire d’habilitation à diriger des recherches, 2008. LIPN, UMR 7030 CNRS, Université Paris 13.
- [BBRDR18] Patrick Baillot, Erika De Benedetti, and Simona Ronchi Della Rocca. Characterizing polynomial and exponential complexity classes in elementary lambda-calculus. *Information and Computation*, 261(1):55–77, 2018.
- [BC92] Stephen Bellantoni and Stephen A. Cook. A new recursion-theoretic characterization of the polytime functions. *Computational Complexity*, 2(2):97–110, 1992.
- [BCJDNM05] Olivier Bournez, Felipe Cucker, Paulin Jacobé De Naurois, and Jean-Yves Marion. Implicit complexity over an arbitrary structure: Sequential and parallel polynomial time. *Journal of Logic and Computation*, 15(1):41–58, 2005.
- [BCJDNM06] Olivier Bournez, Felipe Cucker, Paulin Jacobé De Naurois, and Jean-Yves Marion. Implicit complexity over an arbitrary structure: Quantifier alternations. *Information and Computation*, 204(2):210–230, 2006.
- [BCMT98] Guillaume Bonfante, Adam Cichon, Jean-Yves Marion, and Hélène Touzet. Complexity classes and rewrite systems with polynomial interpretation. In *CSL*, pages 372–384. Springer, 1998.
- [BCMT01] Guillaume Bonfante, Adam Cichon, Jean-Yves Marion, and Hélène Touzet. Algorithms with polynomial interpretation termination proof. *Journal of Functional Programming*, 11(1):33–53, 2001.
- [BCR09] Michael J. Burrell, J. Robin B. Cockett, and Brian F. Redmond. Pola: A language for ptime programming. In *FICS*, pages 7–8. Institute of Cybernetics at Tallinn University of Technology, 2009.
- [BD10] Guillaume Bonfante and Florian Deloup. Complexity invariance of real interpretations. In *TAMC*, pages 139–150. Springer, 2010.
- [BDH15] Guillaume Bonfante, Florian Deloup, and Antoine Henrot. Real or natural number interpretation and their effect on complexity. *Theoretical Computer Science*, 585:25–40, 2015.
- [BDL12] Patrick Baillot and Ugo Dal Lago. Higher-order interpretations and program complexity. In *CSL*, volume 16 of *LIPICs*, pages 62–76. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.
- [BDL16] Patrick Baillot and Ugo Dal Lago. Higher-order interpretations and program complexity. *Information and Computation*, 248:56–81, 2016.
- [BDL18] Flavien Breuvert and Ugo Dal Lago. On intersection types and probabilistic lambda calculi. In *PPDP*, volume 8, pages 1–13. ACM, 2018.
- [BDLGS16] Johannes Borgström, Ugo Dal Lago, Andrew D. Gordon, and Marcin Szymczak. A lambda-calculus foundation for universal probabilistic programming. In *ICFP*, volume 51, pages 33–46. ACM, 2016.

-
- [BDLM12] Patrick Baillot, Ugo Dal Lago, and Jean-Yves Moyen. On quasi-interpretations, blind abstractions and implicit complexity. *Mathematical Structures in Computer Science*, 22(4):549–580, 2012.
- [Bel95] Stephen Bellantoni. Predicative recursion and the polytime hierarchy. In *Feasible Mathematics II*, pages 15–29. Springer, 1995.
- [BG05] Olivier Bournez and Florent Garnier. Proving positive almost-sure termination. In *RTA*, pages 323–337. Springer, 2005.
- [BG06] Olivier Bournez and Florent Garnier. Proving positive almost sure termination under strategies. In *RTA*, pages 357–371. Springer, 2006.
- [BG07] Guillaume Bonfante and Yves Guiraud. Programs as polygraphs: computability and complexity. Technical report, 2007.
- [BG08] Guillaume Bonfante and Yves Guiraud. Intensional properties of polygraphs. *Electronic Notes in Theoretical Computer Science*, 203(1):65–77, 2008.
- [BG18] Patrick Baillot and Alexis Ghyselen. Combining linear logic and size types for implicit complexity. In *CSL*, volume 119 of *LIPICs*, pages 1–21. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018.
- [BGH11] Olivier Bournez, Walid Gomaa, and Emmanuel Hainry. Algebraic characterizations of complexity-theoretic classes of real functions. *International Journal of Unconventional Computing*, 7(5):331–351, 2011.
- [BGP16] Olivier Bournez, Daniel S. Graça, and Amaury Pouly. Polynomial time corresponds to solutions of polynomial ordinary differential equations of polynomial length: The general purpose analog computer and computable analysis are two efficiently equivalent models of computations. In *ICALP*, volume 55 of *LIPICs*, pages 1–15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.
- [BGR08] Gilles Barthe, Benjamin Grégoire, and Colin Riba. Type-based termination with sized products. In *CSL*, pages 493–507. Springer, 2008.
- [BH04] Olivier Bournez and Emmanuel Hainry. Real recursive functions and real extensions of recursive functions. In *MCU*, pages 116–127. Springer, 2004.
- [BHMS04] Lennart Berlinger, Martin Hofmann, Alberto Momigliano, and Olha Shkaravska. Automatic certification of heap consumption. In *LPAR*, pages 347–362. Springer, 2004.
- [Bib77] Kenneth J. Biba. Integrity considerations for secure computer systems. Technical report, Mitre corp rep., 1977.
- [BKMO08] Guillaume Bonfante, Reinhard Kahle, Jean-Yves Marion, and Isabel Oitavem. Recursion schemata for NC^k . In *CSL*, pages 49–63. Springer, 2008.
- [BKV17] Antonio Bucciarelli, Delia Kesner, and Daniel Ventura. Non-idempotent intersection types for the lambda-calculus. *Logic Journal of the IGPL*, 25(4):431–464, 2017.

- [BL11] Alexis Bernadet and Stéphane Lengrand. Complexity of strongly normalising λ -terms via non-idempotent intersection types. In *FoSSaCS*, pages 88–107. Springer, 2011.
- [Blo94] Stephen Bloch. Function-algebraic characterizations of log and polylog parallel time. *Computational Complexity*, 4(2):175–205, 1994.
- [BLO⁺12] Cristina Borralleras, Salvador Lucas, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. Sat modulo linear arithmetic for solving polynomial constraints. *Journal of Automated Reasoning*, 48(1):107–131, 2012.
- [BLP76] David E. Bell and Leonard J. La Padula. Secure computer system: unified exposition and multics interpretation. Technical report, Mitre corp Rep., 1976.
- [Blu67] Manuel Blum. A machine-independent theory of the complexity of recursive functions. *Journal of the ACM*, 14(2):322–336, 1967.
- [BM04] Patrick Baillot and Virgile Mogbil. Soft lambda-calculus: a language for polynomial time computation. In *FoSSaCS*, pages 27–41. Springer, 2004.
- [BM10] Patrick Baillot and Damiano Mazza. Linear logic by levels and bounded time complexity. *Theoretical Computer Science*, 411(2):470–503, 2010.
- [BM12] Aloïs Brunel and Antoine Madet. Indexed realizability for bounded-time programming with references and type fixpoints. In *APLAS*, pages 264–279. Springer, 2012.
- [BMM01] Guillaume Bonfante, Jean-Yves Marion, and Jean-Yves Moyen. On lexicographic termination ordering with space bound certifications. In *PSI*, pages 482–493. Springer, 2001.
- [BMM05] Guillaume Bonfante, Jean-Yves Marion, and Jean-Yves Moyen. Quasi-interpretations and small space bounds. In *RTA*, pages 150–164. Springer, 2005.
- [BMM11] Guillaume Bonfante, Jean-Yves Marion, and Jean-Yves Moyen. Quasi-interpretations a way to control resources. *Theoretical Computer Science*, 412(25):2776–2796, 2011.
- [BMMP05] Guillaume Bonfante, Jean-Yves Marion, Jean-Yves Moyen, and Romain Péchoux. Synthesis of quasi-interpretations. LCC, 2005.
- [BMP06] Guillaume Bonfante, Jean-Yves Marion, and Romain Péchoux. A characterization of alternating log time by first order functional programs. In *LPAR*, pages 90–104. Springer, 2006.
- [BMP07] Guillaume Bonfante, Jean-Yves Marion, and Romain Péchoux. Quasi-interpretation synthesis by decomposition. In *ICTAC*, pages 410–424. Springer, 2007.
- [BN99] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1999.

-
- [BNS00] Stephen Bellantoni, Karl-Heinz Niggl, and Helmut Schwichtenberg. Higher type recursion, ramification and polynomial time. *Annals of Pure and Applied Logic*, 104(1-3):17–30, 2000.
- [Boa14] Peter Van Emde Boas. Machine models and simulations. *Handbook of Theoretical Computer Science*, 1:1–66, 2014.
- [Bon06] Guillaume Bonfante. Some programming languages for Logspace and Ptime . In *AMAST*, pages 66–80. Springer, 2006.
- [Bon11] Guillaume Bonfante. Complexité implicite des calculs : interprétation de programmes. Mémoire d’habilitation à diriger des recherches, 2011. LORIA, UMR 7503, INPL.
- [BP96] Andrew Barber and Gordon Plotkin. Dual intuitionistic linear logic. Technical report, University of Edinburgh, Department of Computer Science, 1996.
- [BP06] Patrick Baillot and Marco Pedicini. An embedding of the BSS model of computation in light affine lambda-calculus. *LCC*, 2006.
- [BRDR15] Erika De Benedetti and Simona Ronchi Della Rocca. Call-by-value, elementary time and intersection types. In *FOPARA*, pages 40–59. Springer, 2015.
- [BSS88] Lenore Blum, Mike Shub, and Steve Smale. On a theory of computation over the real numbers; NP completeness, recursive functions and universal machines. In *FOCS*, pages 387–397. IEEE, 1988.
- [BT04] Patrick Baillot and Kazushige Terui. Light types for polynomial time computation in lambda-calculus. In *LICS*, pages 266–275. IEEE, 2004.
- [BT05] Patrick Baillot and Kazushige Terui. A feasible algorithm for typing in elementary affine logic. In *TLCA*, pages 55–70. Springer, 2005.
- [BT09] Patrick Baillot and Kazushige Terui. Light types for polynomial time computation in lambda calculus. *Information and Computation*, 207(1):41–62, 2009.
- [BT10] Aloïs Brunel and Kazushige Terui. Church \Rightarrow Scott = Ptime : an application of resource sensitive realizability. In *DICE*, volume 23, pages 31–46. Electronic Proceedings in Theoretical Computer Science, 2010.
- [BV97] Ethan Bernstein and Umesh Vazirani. Quantum complexity theory. *SIAM Journal on Computing*, 26(5):1411–1473, 1997.
- [CBGB15] Ranald Clouston, Ales Bizjak, Hans Bugge Grathwohl, and Lars Birkedal. Programming and reasoning with guarded recursion for coinductive types. In *FoS-SaCS*, pages 407–421. Springer, 2015.
- [CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fix-points. In *POPL*, pages 238–252. ACM, 1977.
- [CD96] Wei-Ngan Chin and John Darlington. A higher-order removal method. *Lisp and Symbolic Computation*, 9(4):287–322, 1996.

- [CHS15] Quentin Carbonneaux, Jan Hoffmann, and Zhong Shao. Compositional certified resource bounds. *PLDI*, 50(6):467–478, 2015.
- [CK90] Stephen A. Cook and Bruce M. Kapron. Characterizations of the basic feasible functionals of finite type. In *Feasible mathematics*, pages 71–96. Springer, 1990.
- [CK93] Peter Clote and Jan Krajíček. *Arithmetic, Proof Theory, and Computational Complexity*. Number 23. Oxford University Press, 1993.
- [CK01] Wei-Ngan Chin and Siau-Cheng Khoo. Calculating sized types. *Higher-Order and Symbolic Computation*, 14(2-3):261–300, 2001.
- [CL87] Ahlem Ben Cherifa and Pierre Lescanne. Termination of rewriting systems by polynomial interpretations and its implementation. *Science of Computer Programming*, 9(2):137–159, 1987.
- [CL92] Adam Cichon and Pierre Lescanne. Polynomial interpretations and the complexity of algorithms. In *CADE*, pages 139–147. Springer, 1992.
- [Clo90] Peter Clote. Sequential, machine-independent characterizations of the parallel complexity classes $Alogtime$, AC^k , NC^k and NC . In *Feasible mathematics*, pages 49–69. Springer, 1990.
- [Clo97] Peter Clote. Nondeterministic stack register machines. *Theoretical Computer Science*, 178(1-2):37–76, 1997.
- [Clo99] Peter Clote. Computation models and function algebras. In *Studies in Logic and the Foundations of Mathematics*, volume 140, pages 589–681. Elsevier, 1999.
- [CM00] Adam Cichon and Jean-Yves Marion. The light lexicographic path ordering. Technical report, Loria, 2000.
- [CM01] Paolo Coppola and Simone Martini. Typing lambda terms in elementary logic with linear constraints. In *TLCA*, pages 76–90. Springer, 2001.
- [CMTU05] Evelyne Contejean, Claude Marché, Ana Paula Tomás, and Xavier Urbain. Mechanically proving termination using polynomial interpretations. *Journal of Automated Reasoning*, 34(4):325, 2005.
- [CO07] Manuel Lameiras Campagnolo and Kerry Ojakian. Using approximation to relate computational classes over the reals. In *MCU*, pages 39–61. Springer, 2007.
- [Cob65] Alan Cobham. The intrinsic computational difficulty of functions. *North-Holland Publishing*, 1965.
- [Col89] Loic Colson. About primitive recursive algorithms. In *ICALP*, pages 194–206. Springer, 1989.
- [Con73] Robert L. Constable. Type two computational complexity. In *STOC*, pages 108–121. ACM, 1973.
- [Coo92] Stephen A. Cook. Computability and complexity of higher type functions. In *Logic from Computer Science*, pages 51–72. Springer, 1992.

-
- [Coq94] Thierry Coquand. Infinite objects in type theory. In *TYPES*, volume 806 of *LNCS*, pages 62–78. Springer, 1994.
- [CPR06] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. TERMINATOR: beyond safety. In *CAV*, pages 415–418. Springer, 2006.
- [CR80] Stephen A. Cook and Charles Rackoff. Space lower bounds for maze threadability on restricted machines. *SIAM Journal on Computing*, 9(3):636–652, 1980.
- [CS76] Ashok Chandra and Larry Stockmeyer. Alternation. In *FOCS*, pages 98–108. IEEE, 1976.
- [CS12] Jacek Chrzaszcz and Aleksy Schubert. ML with Ptime complexity guarantees. In *CSL*, volume 16 of *LIPICs*, pages 198–212. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.
- [CS16] Jacek Chrzaszcz and Aleksy Schubert. The role of polymorphism in the characterisation of complexity by soft types. *Information and Computation*, 248:130–149, 2016.
- [CT95] Peter Clote and Gaisi Takeuti. First order bounded arithmetic and small boolean circuit complexity classes. In *Feasible mathematics*, pages 154–218. Springer, 1995.
- [CU93] Stephen A. Cook and Alasdair Urquhart. Functional interpretations of feasibly constructive arithmetic. *Annals of Pure and Applied Logic*, 63(2):103–200, 1993.
- [Dan06] Olivier Danvy. An analytical approach to program as data objects. DSc thesis, 2006. Department of Computer Science, University of Aarhus.
- [DC05] Daniel De Carvalho. Intersection types for light affine lambda calculus. *Electronic Notes in Theoretical Computer Science*, 136:133–152, 2005.
- [DC18] Daniel De Carvalho. Execution time of λ -terms via denotational semantics and intersection types. *Mathematical Structures in Computer Science*, 28(7):1169–1203, 2018.
- [Der79] Nachum Dershowitz. A note on simplification orderings. *Information Processing Letters*, 9(5):212–215, 1979.
- [Der82] Nachum Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17(3):279–301, 1982.
- [Der87] Nachum Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3(1/2):69–116, 1987.
- [Deu85] David Deutsch. Quantum theory, the Church-Turing principle and the universal quantum computer. *Mathematical and Physical Sciences*, 400(1818):97–117, 1985.
- [Dij80] Edsger W. Dijkstra. On the productivity of recursive definitions. EWD749, 1980.
- [DJ03] Vincent Danos and Jean-Baptiste Joinet. Linear logic and elementary time. *Information and Computation*, 183(1):123–137, 2003.

- [DLG11] Ugo Dal Lago and Marco Gaboardi. Linear dependent types and relative completeness. In *LICS*, pages 133–142. IEEE, 2011.
- [DLG19] Ugo Dal Lago and Charles Grellois. Probabilistic termination by monadic affine sized typing. *Transactions on Programming Languages and Systems*, 41(2):10:1–10:65, 2019.
- [DLH05] Ugo Dal Lago and Martin Hofmann. Quantitative models and implicit complexity. In *FSTTCS*, pages 189–200. Springer, 2005.
- [DLH11] Ugo Dal Lago and Martin Hofmann. Realizability models and implicit complexity. *Theoretical Computer Science*, 412(20):2029–2047, 2011.
- [DLM09] Ugo Dal Lago and Simone Martini. On constructor rewrite systems and the lambda-calculus. In *ICALP*, pages 163–174. Springer, 2009.
- [DLMS10] Ugo Dal Lago, Simone Martini, and Davide Sangiorgi. Light logics and higher-order processes. In *EXPRESS*, volume 41, pages 46–60. Electronic Proceedings in Theoretical Computer Science, 2010.
- [DLMS16] Ugo Dal Lago, Simone Martini, and Davide Sangiorgi. Light logics and higher-order processes. *Mathematical Structures in Computer Science*, 26(6):969–992, 2016.
- [DLMZ10] Ugo Dal Lago, Andrea Masini, and Margherita Zorzi. Quantum implicit computational complexity. *Theoretical Computer Science*, 411(2):377–409, 2010.
- [DLP14] Ugo Dal Lago and Barbara Petit. Linear dependent types in a call-by-value scenario. *Science of Computer Programming*, 84:77–100, 2014.
- [DLR15] Norman Danner, Daniel R. Licata, and R. Ramyaa. Denotational cost semantics for functional languages with inductive types. In *ICFP*, volume 50, pages 140–151. ACM, 2015.
- [DLS10] Ugo Dal Lago and Ulrich Schöpp. Functional programming in sublinear space. In *ESOP*, pages 205–225. Springer, 2010.
- [DLS16] Ugo Dal Lago and Ulrich Schöpp. Computation by interaction for space-bounded functional programming. *Information and Computation*, 248:150–194, 2016.
- [DLT15] Ugo Dal Lago and Paolo Parisen Toldin. A higher-order characterization of probabilistic polynomial time. *Information and Computation*, 241:114–141, 2015.
- [EGH⁺10] Jörg Endrullis, Clemens Grabmayer, Dimitri Hendriks, Ariya Ishihara, and Jan Willem Klop. Productivity of stream definitions. *Theoretical Computer Science*, 411(4-5):765–782, 2010.
- [ER03] Thomas Ehrhard and Laurent Regnier. The differential lambda-calculus. *Theoretical Computer Science*, 309(1-3):1–41, 2003.
- [ER08] Thomas Ehrhard and Laurent Regnier. Uniformity and the taylor expansion of ordinary lambda-terms. *Theoretical Computer Science*, 403(2-3):347–372, 2008.

-
- [EWZ08] Jörg Endrullis, Johannes Waldmann, and Hans Zantema. Matrix interpretations for proving termination of term rewriting. *Journal of Automated Reasoning*, 40(2-3):195–220, 2008.
- [Fag74] Ronald Fagin. Generalized first-order spectra and polynomial-time recognizable sets. *Complexity of computation*, 7:43–73, 1974.
- [Fag19] Claudia Faggian. Probabilistic rewriting: Normalization, termination, and unique normal forms. In *FSCD*, volume 131 of *LIPICs*, pages 1–25. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2019.
- [FGM⁺07] Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René Thiemann, and Harald Zankl. SAT solving for termination analysis with polynomial interpretations. In *SAT*, pages 340–354. Springer, 2007.
- [FHHP10] Hugo Férée, Emmanuel Hainry, Mathieu Hoyrup, and Romain Péchoux. Interpretation of stream programs: Characterizing type 2 polynomial time complexity. In *ISAAC*, pages 291–303. Springer, 2010.
- [FHHP15] Hugo Férée, Emmanuel Hainry, Mathieu Hoyrup, and Romain Péchoux. Characterizing polynomial time complexity of stream programs using interpretations. *Theoretical Computer Science*, 585:41–54, 2015.
- [FHM⁺18] Hugo Férée, Samuel Hym, Micaela Mayero, Jean-Yves Moyen, and David Nowak. Formal proof of polynomial-time complexity with quasi-interpretations. In *CPP*, pages 146–157. ACM, 2018.
- [FRDR19] Claudia Faggian and Simona Ronchi Della Rocca. Lambda calculus and probabilistic computation. In *LICS*, pages 1–13. IEEE, 2019.
- [GAB⁺17] Jürgen Giesl, Cornelius Aschermann, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Jera Hensel, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, et al. Analyzing program termination and complexity automatically with AProVE. *Journal of Automated Reasoning*, 58(1):3–31, 2017.
- [Geu88] Oliver Geupel. Terminationsbeweise bei Termersetzungssystemen. diplomarbeit, 1988. Technische Universität Dresden, Sektion Mathematik.
- [Gie95] Jürgen Giesl. Generating polynomial orderings for termination proofs. In *RTA*, pages 426–431. Springer, 1995.
- [Gir72] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l’arithmétique d’ordre supérieur*. PhD thesis, Université Paris VII, 1972.
- [Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–101, 1987.
- [Gir98] Jean-Yves Girard. Light linear logic. *Information and Computation*, 143(2):175–204, 1998.
- [GM16] Stéphane Gimenez and Georg Moser. The complexity of interaction. In *POPL*, volume 51, pages 243–255. ACM, 2016.

- [GMC09] Sumit Gulwani, Krishna K Mehra, and Trishul Chilimbi. SPEED: precise and efficient static estimation of program computational complexity. In *POPL*, volume 44, pages 127–139. ACM, 2009.
- [GMRDR08a] Marco Gaboardi, Jean-Yves Marion, and Simona Ronchi Della Rocca. A logical account of PSPACE. volume 43, pages 121–131. ACM, 2008.
- [GMRDR08b] Marco Gaboardi, Jean-Yves Marion, and Simona Ronchi Della Rocca. Soft linear logic and polynomial complexity classes. *Electronic Notes in Theoretical Computer Science*, 205:67–87, 2008.
- [Gna07] Isabelle Gnaedig. Induction for positive almost sure termination. In *PPDP*, pages 167–178. ACM, 2007.
- [Göd31] Kurt Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für mathematik und physik*, 38(1):173–198, 1931.
- [Goe92] Andreas Goerdt. Characterizing complexity classes by general recursive definitions in higher types. *Information and Computation*, 101(2):202–218, 1992.
- [GP09a] Marco Gaboardi and Romain Péchoux. Global and local space properties of stream programs. In *FOPARA*, pages 51–66. Springer, 2009.
- [GP09b] Marco Gaboardi and Romain Péchoux. Upper bounds on stream I/O using semantic interpretations. In *CSL*, pages 271–286. Springer, 2009.
- [GP15a] Marco Gaboardi and Romain Péchoux. Algebras and coalgebras in the light affine lambda calculus. In *ICFP*, volume 50, pages 114–126. ACM, 2015.
- [GP15b] Marco Gaboardi and Romain Péchoux. On bounding space usage of streams using interpretation analysis. *Science of Computer Programming*, 111:395–425, 2015.
- [Grä91] Erich Grädel. The expressive power of second order Horn logic. In *STACSs*, pages 466–477. Springer, 1991.
- [Gra94] Bernhard Gramlich. Generalized sufficient conditions for modular termination of rewriting. *Applicable Algebra in Engineering, Communication and Computing*, 5(3-4):131–158, 1994.
- [GRDR07] Marco Gaboardi and Simona Ronchi Della Rocca. A soft type assignment system for λ -calculus. In *CSL*, pages 253–267. Springer, 2007.
- [GRDR08] Marco Gaboardi and Simona Ronchi Della Rocca. Type inference for a polynomial lambda calculus. In *TYPES*, pages 136–152. Springer, 2008.
- [Gro96] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *STOC*, pages 212–219. ACM, 1996.
- [Grz53] Andrzej Grzegorzcyk. Some classes of recursive functions. Technical report, Instytut Matematyczny Polskiej Akademi Nauk (Warszawa), 1953.

-
- [GSS92] Jean-Yves Girard, Andre Scedrov, and Philip Scott. Bounded linear logic: a modular approach to polynomial-time computability. *Theoretical Computer Science*, 97(1):1–66, 1992.
- [Gul09] Sumit Gulwani. SPEED: Symbolic complexity bound analysis. In *CAV*, pages 51–62. Springer, 2009.
- [Gur83] Yuri Gurevich. Algebras of feasible functions. In *FOCS*, pages 210–214. IEEE, 1983.
- [Hag87] Tatsuya Hagino. A typed lambda calculus with categorical type constructors. In *CTCS*, volume 283, pages 140–57. Springer, 1987.
- [HAH11] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate amortized resource analysis. In *POPL*, volume 46, pages 357–370. ACM, 2011.
- [HAH12] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Resource aware ML. In *CAV*, pages 781–786. Springer, 2012.
- [Haj79] Petr Hajek. Arithmetical hierarchy and complexity of computation. *Theoretical Computer Science*, 8(2):227–237, 1979.
- [HH10] Jan Hoffmann and Martin Hofmann. Amortized resource analysis with polynomial potential. In *ESOP*, pages 287–306. Springer, 2010.
- [HJ03] Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. In *POPL*, volume 38, pages 185–197. ACM, 2003.
- [HJ06] Martin Hofmann and Steffen Jost. Type-based amortised heap-space analysis. In *ESOP*, pages 22–37. Springer, 2006.
- [HJP10] Matthias Heizmann, Neil Jones, and Andreas Podelski. Size-change termination and transition invariants. In *SAS*, pages 22–50. Springer, 2010.
- [HM08] Nao Hirokawa and Georg Moser. Automated complexity analysis based on the dependency pair method. In *IJCAR*, pages 364–379. Springer, 2008.
- [HM14a] Nao Hirokawa and Georg Moser. Automated complexity analysis based on context-sensitive rewriting. In *TLCA and RTA*, pages 257–271. Springer, 2014.
- [HM14b] Martin Hofmann and Georg Moser. Amortised resource analysis and typed polynomial interpretations. In *TLCA and RTA*, pages 272–286. Springer, 2014.
- [HM15] Martin Hofmann and Georg Moser. Multivariate amortised resource analysis for term rewrite systems. In *TLCA*, volume 38 of *LIPICs*, pages 241–256. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.
- [HMP13] Emmanuel Hainry, Jean-Yves Marion, and Romain Péchoux. Type-based complexity analysis for fork processes. In *FoSSaCS*, pages 305–320. Springer, 2013.
- [Hof92] Dieter Hofbauer. Termination proofs by multiset path orderings imply primitive recursive derivation lengths. *Theoretical Computer Science*, 105(1):129–140, 1992.

- [Hof99] Martin Hofmann. Linear types and non-size-increasing polynomial time computation. In *LICS*, pages 464–473. IEEE, 1999.
- [Hof00] Martin Hofmann. Safe recursion with higher types and BCK-algebra. *Annals of Pure and Applied Logic*, 104(1-3):113–166, 2000.
- [Hof01] Dieter Hofbauer. Termination proofs by context-dependent interpretations. In *RTA*, pages 108–121. Springer, 2001.
- [Hof02] Martin Hofmann. The strength of non-size increasing computation. *POPL*, 37(1):260–269, 2002.
- [Hof03] Martin Hofmann. Linear types and non-size-increasing polynomial time computation. *Information and Computation*, 183(1):57–85, 2003.
- [HP15] Emmanuel Hainry and Romain Péchoux. Objects in polynomial time. In *APLAS*, pages 387–404. Springer, 2015.
- [HP17] Emmanuel Hainry and Romain Péchoux. Higher order interpretation for higher order complexity. In *LPAR, EPIC*, pages 269–285. easychair, 2017.
- [HP18] Emmanuel Hainry and Romain Péchoux. A type-based complexity analysis of object oriented programs. *Information and Computation*, 261(1):78–115, 2018.
- [HPS96] John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In *POPL*, pages 410–423. ACM, 1996.
- [HR09] Martin Hofmann and Dulma Rodriguez. Efficient type-checking for amortised heap-space analysis. In *CSL*, pages 317–331. Springer, 2009.
- [HR13] Martin Hofmann and Dulma Rodriguez. Automatic type inference for amortised heap-space analysis. In *ESOP*, pages 593–613. Springer, 2013.
- [HRS13] Martin Hofmann, Ramyaa Ramyaa, and Ulrich Schöpp. Pure pointer programs and tree isomorphism. In *FoSSaCS*, pages 321–336. Springer, 2013.
- [HS10] Martin Hofmann and Ulrich Schöpp. Pure pointer programs with iteration. *Transactions on Computational Logic*, 11(4):26, 2010.
- [HS15] Jan Hoffmann and Zhong Shao. Automatic static cost analysis for parallel programs. In *ESOP*, pages 132–157. Springer, 2015.
- [HW06] Dieter Hofbauer and Johannes Waldmann. Termination of string rewriting with matrix interpretations. In *RTA*, pages 328–342. Springer, 2006.
- [IKR02] Robert J. Irwin, Bruce M. Kapron, and James S. Royer. On characterizations of the basic feasible functionals, Part II. Technical report, Syracuse University, 2002.
- [Imm86] Neil Immerman. Relational queries computable in polynomial time. *Information and Control*, 68(1-3):86–104, 1986.
- [Imm12] Neil Immerman. *Descriptive Complexity*. Springer Science & Business Media, 2012.

-
- [IPW01] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.
- [IRK01] Robert J. Irwin, James S. Royer, and Bruce M. Kapron. On characterizations of the basic feasible functionals, Part I. *Journal of Functional Programming*, 11(1):117–153, 2001.
- [JDN19] Paulin Jacobé De Naurois. Pointers in recursion: Exploring the tropics. In *FSCD*, volume 131 of *LIPICs*, pages 1–18. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2019.
- [JHLH10] Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann. Static determination of quantitative resource usage for higher-order programs. In *POPL*, volume 45, pages 223–236. ACM, 2010.
- [JK05] Neil Jones and Lars Kristiansen. The flow of data and the complexity of algorithms. In *CIE*, pages 263–274. Springer, 2005.
- [JK09] Neil Jones and Lars Kristiansen. A flow calculus of mwp-bounds for complexity analysis. *Transactions on Computational Logic*, 10(4):28, 2009.
- [Jon99] Neil Jones. **Logspace** and **Ptime** characterized by programming languages. *Theoretical Computer Science*, 228(1-2):151–174, 1999.
- [Jon01] Neil Jones. The expressive power of higher-order types or, life without CONS. *Journal of Functional Programming*, 11(1):55–94, 2001.
- [JR97] Bart Jacobs and Jan Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:222–259, 1997.
- [K⁺01] Jan Willem Klop et al. *Term Rewriting Systems*. Cambridge University Press, 2001.
- [Kal43] László Kalmár. Egyszerű példa eldönthetetlen aritmetikai problémára. *Mate és fizikai lapok*, 50:1–23, 1943.
- [Kam80] Samuel Kamin. Attempts for generalizing the recursive path orderings. Technical report, Report, Dept. of Computer Science, University of Illinois, 1980.
- [KC91] Bruce M. Kapron and Stephen A. Cook. A new characterization of Mehlhorn’s polynomial time functionals. In *FOCS*, pages 342–347. IEEE, 1991.
- [KC96] Bruce M. Kapron and Steven A. Cook. A new characterization of type-2 feasibility. *SIAM Journal on Computing*, 25(1):117–132, 1996.
- [KDLG19] Naoki Kobayashi, Ugo Dal Lago, and Charles Grellois. On the termination problem for probabilistic higher-order recursive programs. In *LICS*, pages 1–14. IEEE, 2019.
- [KKMO16] Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. Weakest precondition reasoning for expected runtimes of probabilistic programs. In *ESOP*, pages 364–389. Springer, 2016.

- [Kle35] Stephen Kleene. A theory of positive integers in formal logic. Part I. *American Journal of Mathematics*, 57(1):153–173, 1935.
- [Kle36] Stephen Kleene. General recursive functions of natural numbers. *Mathematische annalen*, 112(1):727–742, 1936.
- [KN85] Mukkai Krishnamoorthy and Paliath Narendran. On recursive path ordering. *Theoretical Computer Science*, 40:323–328, 1985.
- [KN04] Lars Kristiansen and Karl-Heinz Niggl. On the computational complexity of imperative programming languages. *Theoretical Computer Science*, 318(1-2):139–161, 2004.
- [Ko91] Ker-I Ko. *Complexity Theory of Real Functions*. Birkhäuser, 1991.
- [KO92] Masahito Kurihara and Azuma Ohuchi. Modularity of simple termination of term rewriting systems with shared constructors. *Theoretical Computer Science*, 103(2):273–282, 1992.
- [Koz79] Dexter Kozen. Semantics of probabilistic programs. In *SFCS*, pages 101–114. IEEE, 1979.
- [KP84] Brian W. Kernighan and Rob Pike. *The Unix Programming Environment*. Prentice Hall, 1984.
- [KS18] Bruce M. Kapron and Florian Steinberg. Type-two polynomial-time and restricted lookahead. In *LICS*, pages 579–588. IEEE, 2018.
- [KS19] Bruce M. Kapron and Florian Steinberg. Type-two iteration with bounded query revision. In *DICE-FOPARA*, volume 298, pages 61–74. Electronic Proceedings in Theoretical Computer Science, 2019.
- [KSvG⁺12] Rody Kersten, Olha Shkaravska, Bernard van Gastel, Manuel Montenegro, and Marko C. J. D. van Eekelen. Making resource analysis practical for real-time Java. In *JTRES*, pages 135–144. ACM, 2012.
- [KV03] Lars Kristiansen and Paul Voda. Complexity classes and fragments of C. *Information Processing Letters*, 88(5):213–218, 2003.
- [KV17] Delia Kesner and Pierre Vial. Types as resources for classical natural deduction. In *FSCD*, volume 84 of *LIPICs*, pages 1–17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.
- [Laf04] Yves Lafont. Soft linear logic and polynomial time. *Theoretical Computer Science*, 318(1-2):163–180, 2004.
- [Lan79] Dallas Lankford. On proving term rewriting systems are Noetherien. Technical report, Department of Mathematics, Louisiana Technical University, 1979.
- [Lau88] Clemens Lautemann. A note on polynomial interpretation. *Bulletin of the EATCS*, 36:129–130, 1988.
- [Lei95] Daniel Leivant. Ramified recurrence and computational complexity I: Word recurrence and poly-time. In *Feasible mathematics*, pages 320–343. Springer, 1995.

-
- [Lei98] Daniel Leivant. A characterization of NC by tree recurrence. In *FOCS*, pages 716–724. IEEE, 1998.
- [LJBA01] Chin Soon Lee, Neil Jones, and Amir Ben-Amram. The size-change principle for program termination. In *POPL*, volume 36, pages 81–92. ACM, 2001.
- [LM93] Daniel Leivant and Jean-Yves Marion. Lambda calculus characterizations of poly-time. In *TLCA*, pages 274–288. Springer, 1993.
- [LM94] Daniel Leivant and Jean-Yves Marion. Ramified recurrence and computational complexity II: substitution and poly-space. In *CSL*, pages 486–500. Springer, 1994.
- [LM00] Daniel Leivant and Jean-Yves Marion. A characterization of alternating log time by ramified recurrence. *Theoretical Computer Science*, 236(1-2):193–208, 2000.
- [LM13] Daniel Leivant and Jean-Yves Marion. Evolving graph-structures and their implicit computational complexity. In *ICALP*, pages 349–360. Springer, 2013.
- [LTDF06] Olivier Laurent and Lorenzo Tortora De Falco. Obsessional cliques: a semantic characterization of bounded time complexity. In *LICS*, pages 179–188. IEEE, 2006.
- [Luc05] Salvador Lucas. Polynomials over the reals in proofs of termination: from theory to practice. *RAIRO-Theoretical Informatics and Applications*, 39(3):547–586, 2005.
- [Luc07] Salvador Lucas. Practical use of polynomials over the reals in proofs of termination. In *PPDP*, pages 39–50. ACM, 2007.
- [MA11] Antoine Madet and Roberto M. Amadio. An elementary affine λ -calculus with multithreading and side effects. In *TLCA*, pages 138–152. Springer, 2011.
- [Mad12] Antoine Madet. A polynomial time λ -calculus with multithreading and side effects. In *PPDP*, pages 55–66. ACM, 2012.
- [Mar11] Jean-Yves Marion. A type system for complexity flow analysis. In *LICS*, pages 123–132. IEEE, 2011.
- [Mau03] François Maurel. Nondeterministic light logics and NP-time. In *TLCA*, pages 241–255. Springer, 2003.
- [Maz12] Damiano Mazza. An infinitary affine lambda-calculus isomorphic to the full lambda-calculus. In *LICS*, pages 471–480. IEEE, 2012.
- [Maz14] Damiano Mazza. Non-uniform polytime computation in the infinitary affine lambda-calculus. In *ICALP*, pages 305–317. Springer, 2014.
- [Maz15] Damiano Mazza. Simple parsimonious types and logarithmic space. In *CSL*, volume 41 of *LIPICs*, pages 24–40. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.
- [Meh76] Kurt Mehlhorn. Polynomial and abstract subrecursive classes. *Journal of Computer and System Sciences*, 12(2):147–178, 1976.

- [MM00] Jean-Yves Marion and Jean-Yves Moyen. Efficient first order functional program interpreter with time bound certifications. In *LPAR*, pages 25–42. Springer, 2000.
- [MN70] Zohar Manna and Steven Ness. On the termination of markov algorithms. In *HICSS*, pages 789–792. IEEE, 1970.
- [MO04] Andrzej Murawski and Luke Ong. On an interpretation of safe recursion in light affine logic. *Theoretical Computer Science*, 318(1-2):197–223, 2004.
- [Moy09] Jean-Yves Moyen. Resource control graphs. *Transactions on Computational Logic*, 10(4):29, 2009.
- [MP06] Jean-Yves Marion and Romain Péchoux. Resource analysis by sup-interpretation. In *FLOPS*, pages 163–176. Springer, 2006.
- [MP08a] Jean-Yves Marion and Romain Péchoux. Analyzing the implicit computational complexity of object-oriented programs. In *FSTTCS*, volume 2 of *LIPICs*, pages 316–327. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2008.
- [MP08b] Jean-Yves Marion and Romain Péchoux. A characterization of NC^k by first order functional programs. In *TAMC*, pages 136–147. Springer, 2008.
- [MP08c] Jean-Yves Marion and Romain Péchoux. Characterizations of polynomial complexity classes with a better intensionality. In *PPDP*, pages 79–88. ACM, 2008.
- [MP09] Jean-Yves Marion and Romain Péchoux. Sup-interpretations, a semantic method for static analysis of program resources. *Transactions on Computational Logic*, 10(4):27, 2009.
- [MP14] Jean-Yves Marion and Romain Péchoux. Complexity information flow in a multi-threaded imperative language. In *TAMC*, pages 124–140. Springer, 2014.
- [MS08] Georg Moser and Andreas Schnabl. Proving quadratic derivational complexities using context dependent interpretations. In *RTA*, pages 276–290. Springer, 2008.
- [MS09] Georg Moser and Andreas Schnabl. The derivational complexity induced by the dependency pair method. In *RTA*, pages 255–269. Springer, 2009.
- [MS11] Georg Moser and Andreas Schnabl. Termination Proofs in the Dependency Pair Framework May Induce Multiple Recursive Derivational Complexity. In *RTA*, volume 10 of *LIPICs*, pages 235–250. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011.
- [MSW08] Georg Moser, Andreas Schnabl, and Johannes Waldmann. Complexity analysis of term rewriting based on matrix and context dependent interpretations. In *RTA*, volume 2 of *LIPICs*, pages 304–315. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2008.
- [MT15] Damiano Mazza and Kazushige Terui. Parsimonious types and non-uniform computation. In *ICALP*, pages 350–361. Springer, 2015.
- [Mül74] Helmut Müller. *Klassifizierungen der primitiv-rekursiven Funktionen*. PhD thesis, Verlag nicht ermittelbar, 1974.

-
- [NCH18] Van Chan Ngo, Quentin Carbonneaux, and Jan Hoffmann. Bounded expectations: resource analysis for probabilistic programs. In *PLDI*, volume 53, pages 496–512. ACM, 2018.
- [NEG13] Lars Noschinski, Fabian Emmes, and Jürgen Giesl. Analyzing innermost runtime complexity of term rewriting by dependency pairs. *Journal of Automated Reasoning*, 51(1):27–56, 2013.
- [Nig00] Karl-Heinz Niggl. The μ -measure as a tool for classifying computational complexity. *Archive for Mathematical Logic*, 39(7):515–539, 2000.
- [NM10] Friedrich Neurauder and Aart Middeldorp. Polynomial interpretations over the reals do not subsume polynomial interpretations over the integers. In *RTA*, volume 6 of *LIPICs*, pages 243–258. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010.
- [NW06] Karl-Heinz Niggl and Henning Wunderlich. Certifying polynomial time and linear/polynomial space for imperative programs. *SIAM Journal on Computing*, 35(5):1122–1147, 2006.
- [NZM10] Friedrich Neurauder, Harald Zankl, and Aart Middeldorp. Revisiting matrix interpretations for polynomial derivational complexity of term rewriting. In *LPAR*, pages 550–564. Springer, 2010.
- [Oit01] Isabel Oitavem. Implicit characterizations of PSPACE. In *PTCS*, pages 170–190. Springer, 2001.
- [Pap03] Christos Papadimitriou. *Computational Complexity*. John Wiley and Sons Ltd., 2003.
- [Péc13] Romain Péchoux. Synthesis of sup-interpretations: a survey. *Theoretical Computer Science*, 467:30–52, 2013.
- [Per08] Simon Perdrix. Quantum entanglement analysis based on abstract interpretation. In *SAS*, pages 270–282. Springer, 2008.
- [Per18] Matthieu Perrinel. Paths-based criteria and application to linear logic subsystems characterizing polynomial time. *Information and Computation*, 261:23–54, 2018.
- [Pét67] Rózsa Péter. *Recursive Functions*. Academic Press, 1967.
- [Pie91] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists*. MIT press, 1991.
- [Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT press, 2002.
- [Pit98] François Pitt. A quantifier-free theory based on a string algebra for NC^1 . In *Proof complexity and feasible arithmetics*, DIMACS, pages 229–252. AMS, 1998.
- [Pla78] David Plaisted. A recursively defined ordering for proving termination of term rewriting systems. Technical Report 943, Department of Computer Science, University of Illinois at Urbana-Champaign, 1978.

- [PR04] Andreas Podelski and Andrey Rybalchenko. Transition invariants. In *LICS*, pages 32–41. IEEE, 2004.
- [Red07] Brian F. Redmond. Multiplexor categories and models of soft linear logic. In *LFCS*, pages 472–485. Springer, 2007.
- [Rei08] Omer Reingold. Undirected connectivity in log-space. *Journal of the ACM*, 55(4):17, 2008.
- [Rey72] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM annual conference*, volume 2, pages 717–740. ACM, 1972.
- [Rey74] John C. Reynolds. Towards a theory of type structure. In *Programming Symposium*, pages 408–425. Springer, 1974.
- [Rit63] Robert Ritchie. Classes of predictably computable functions. *Transactions of the American Mathematical Society*, 106(1):139–173, 1963.
- [Ros87] Harvey Rose. *Subrecursion: Functions and Hierarchies*. Oxford University Press, 1987.
- [Rov99] Luca Roversi. A p-time completeness proof for light logics. In *CSL*, pages 469–483. Springer, 1999.
- [Ruz81] Walter L. Ruzzo. On uniform circuit complexity. *Journal of Computer and System Sciences*, 22(3):365–383, 1981.
- [San90] David Sands. Complexity analysis for a lazy higher-order language. In *ESOP*, pages 361–376. Springer, 1990.
- [San91] David Sands. Time analysis, cost equivalence and program refinement. In *FSTTCS*, pages 25–39. Springer, 1991.
- [Sav98] John E. Savage. *Models of Computation*, volume 136. Addison-Wesley Reading, MA, 1998.
- [Saz80] Vladimir Sazonov. Polynomial computability and recursivity in finite domains. *Elektronische Informationsverarbeitung und Kybernetik*, 16(7):319–323, 1980.
- [Sch69] Helmut Schwichtenberg. Rekursionszahlen und die Grzegorzcyk-Hierarchie. *Archiv für mathematische Logik und Grundlagenforschung*, 12(1-2):85–97, 1969.
- [Sch07] Ulrich Schöpp. Stratified bounded affine logic for logarithmic space. In *LICS*, volume 7, pages 411–420. IEEE, 2007.
- [Sel04] Peter Selinger. Towards a quantum programming language. *Mathematical Structures in Computer Science*, 14(4):527–586, 2004.
- [Sev17] Paula Severi. A light modality for recursion. In *FoSSaCS*, pages 499–516. Springer, 2017.
- [Sho97] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, 1997.

-
- [Sij89] Ben A. Sijtsma. On the productivity of recursive list definitions. *Transactions on Programming Languages and Systems*, 11(4):633–649, 1989.
- [Sip06] Michael Sipser. *Introduction to the Theory of Computation*. Thomson Course Technology, Boston, 2006.
- [SKvE10] Olha Shkaravska, Rody Kersten, and Marko C. J. D. van Eekelen. Test-based inference of polynomial loop-bound functions. In *PPDP*, pages 99–108. ACM, 2010.
- [SS05] Andrei Sabelfeld and David Sands. Dimensions and principles of declassification. In *CSFW*, pages 255–269. IEEE, 2005.
- [Ste92] Joachim Steinbach. Proving polynomials positive. In *FSTTCS*, pages 191–202. Springer, 1992.
- [SV98] Geoffrey Smith and Dennis Volpano. Secure information flow in a multi-threaded imperative language. In *POPL*, pages 355–364. ACM, 1998.
- [SV06] Peter Selinger and Benoit Valiron. A lambda calculus for quantum computation with classical control. *Mathematical Structures in Computer Science*, 16(3):527–552, 2006.
- [SvET13] Olha Shkaravska, Marko C. J. D. van Eekelen, and Alejandro Tamalet. Collected size semantics for strict functional programs over general polymorphic lists. In *FOPARA*, pages 143–159. Springer, 2013.
- [SvEvK09] Olha Shkaravska, Marko C. J. D. van Eekelen, and Ron van Kesteren. Polynomial size analysis of first-order shapely functions. *Logical Methods in Computer Science*, 5(2), 2009.
- [SvKvE07] Olha Shkaravska, Ron van Kesteren, and Marko C. J. D. van Eekelen. Polynomial size analysis of first-order functions. In *TLCA*, pages 351–365. Springer, 2007.
- [SW03] Davide Sangiorgi and David Walker. *The Pi-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2003.
- [Tar51] Alfred Tarski. *A Decision Method for Elementary Algebra and Geometry*. University of California Press, 1951.
- [Ter01] Kazushige Terui. Light affine lambda calculus and polytime strong normalization. In *LICS*, pages 209–220. IEEE, 2001.
- [Var82] Moshe Vardi. The complexity of relational query languages. In *STOC*, pages 137–146. ACM, 1982.
- [Vas08] Pedro Vasconcelos. *Space cost analysis using sized types*. PhD thesis, University of St Andrews, 2008.
- [VIS96] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2/3):167–188, 1996.

- [VJFH15] Pedro Vasconcelos, Steffen Jost, Mário Florido, and Kevin Hammond. Type-based allocation analysis for co-recursion in lazy functional languages. In *ESOP*, pages 787–811. Springer, 2015.
- [VW96] Heribert Vollmer and Klaus Wagner. Recursion theoretic characterizations of complexity classes of counting functions. *Theoretical Computer Science*, 163(1-2):245–258, 1996.
- [Wal10] Johannes Waldmann. Polynomially bounded matrix interpretations. In *RTA*, volume 6 of *LIPICs*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010.
- [Wal15] Johannes Waldmann. Matrix interpretations on polyhedral domains. In *RTA*, volume 36 of *LIPICs*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.
- [Wei95] Andreas Weiermann. Termination proofs for term rewriting systems by lexicographic path orderings imply multiply recursive derivation lengths. *Theoretical Computer Science*, 139(1-2):355–362, 1995.
- [Wei00] Klaus Weihrauch. *Computable Analysis: An Introduction*. Springer, 2000.
- [Win93] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT press, 1993.
- [Wra93] Gavin C. Wraith. A note on categorical datatypes. In *CTCS*, volume 389, pages 118–127. Springer, 1993.
- [Yam01] Toshiyuki Yamada. Confluence and termination of simply typed term rewriting systems. In *RTA*, pages 338–352. Springer, 2001.
- [Yam18] Tomoyuki Yamakami. A schematic definition of quantum polynomial time computability. *arXiv preprint arXiv:1802.02336*, 2018.
- [Zoo] Complexity Zoo. https://complexityzoo.uwaterloo.ca/Complexity_Zoo.

Une sélection de 5 publications

1 Résumé des publications

Les cinq publications, fournies en copie dans les sections qui suivent, s'inscrivent toutes dans ma thématique principale sur la complexité implicite. Un bref résumé de ces différentes contributions est fourni ci-dessous.

1. Romain Péchoux. Synthesis of sup-interpretations: A survey. *Theoretical Computer Science*, 41 pages, 2013.
 - Cet article est un “survey” des résultats obtenus durant mon doctorat sur l'utilisation de la notion d'interprétation polynomiale (habituellement utilisée pour démontrer la terminaison des systèmes de réécriture) afin d'analyser la complexité des programmes. Il présente deux extensions, les quasi-interprétations et sup-interprétations, permettant d'augmenter l'expressivité des programmes analysés. Il démontre que de tels outils permettent des caractérisations du temps polynomial et de l'espace polynomial. Enfin, il étudie la complexité et la décidabilité du problème de l'inférence d'une interprétation pour un programme donné (indécidable en général, exponentiel sur l'espace des polynômes à coefficients réels et NP-complet sur l'espace des fonctions linéaires et discrètes).
2. Marco Gaboardi and Romain Péchoux. Algebras and coalgebras in the light affine lambda calculus. In *the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015. Proceedings*, 12 pages, ACM, 2015.
 - Cet article s'appuie sur les travaux de Wraith et Wadler qui ont démontré que le système F peut être étendu par des types inductifs et coinductifs sans perdre sa propriété de normalisation forte pour démontrer que le lambda calcul typé par logique affine et légère (LALC) peut lui aussi être étendu par des types inductifs et coinductifs sans perdre sa propriété de normalisation en temps polynomial. Ainsi ce papier contient un résultat théorique (les notions d'algèbre et coalgèbre de la théorie des catégories peuvent aussi être adaptées aux logiques affines et légères) et un résultat pratique (l'expressivité des langages fonctionnels est augmentée : on peut désormais programmer des programmes sur des listes, arbres, streams infinis, ... tout en garantissant que le programme termine en temps polynomial).
3. Hugo Férée, Emmanuel Hainry, Mathieu Hoyrup, and Romain Péchoux. Characterizing polynomial time complexity of stream programs using interpretations. *Theoretical Computer Science*, 24 pages, 2015.

- Cet article est une première adaptation de la notion d'interprétation à l'ordre supérieur. Pour ce faire, il considère des systèmes de réécriture sur les streams, des suites infinies d'entiers. Un nombre réel x est alors vu comme un stream de rationnels (la suite de Cauchy qui converge vers x). Ce papier fournit une première caractérisation de la classe BFF_2 introduite par Kapron et Cook (l'équivalent du temps polynomial à l'ordre 2). Par ailleurs, il établit une caractérisation analogue en analyse récursive.
4. Marco Gaboardi and Romain Péchoux. On Bounding Space Usage of Streams Using Interpretation Analysis. *Science of Computer Programming*, 46 pages, 2015.
- Cet article développe des analyses statiques à base d'interprétation permettant de prouver des propriétés de complexité sur des langages fonctionnels utilisant des streams. Par exemple, il permet de prouver des bornes supérieures en taille sur les éléments produits en sortie d'un stream en fonction de la taille des éléments en entrée. Par ailleurs, il permet de fournir des bornes sur le nombre d'éléments devant être lus en entrée afin de produire un élément en sortie. Les applications pratiques de ces résultats sont très claires : ils permettent de connaître avant exécution et par analyse statique des bornes sur l'espace à allouer en mémoire afin de stocker des éléments et sur la taille nécessaire du buffer afin d'éviter des overflows en mémoire.
5. Emmanuel Hainry and Romain Péchoux. A type-based complexity analysis of Object Oriented programs. *Information and Computation*, 60 pages, 2018.
- Ce papier décrit un résultat novateur : l'adaptation de la technique du Tiering développée par Bellantoni et Cook en 1992 afin de caractériser la classe des fonctions calculables en temps polynomial (dans un langage fonctionnel) à un langage Orienté Objet. Il permet ainsi de fournir une caractérisation théorique de la classe de complexité FP mais aussi des bornes pratiques sur les tailles du tas et de la pile des programmes OO typables dans ce formalisme. Des applications pratiques de ce résultat ont été obtenues (Logiciel: COMPLEXITYPARSER développé en collaboration avec Emmanuel Hainry et Olivier Zeyen) et soulignent son fort potentiel.

2 Synthesis of sup-interpretations: A survey

Synthesis of sup-interpretations: a survey

Romain Péchoux

► **To cite this version:**

Romain Péchoux. Synthesis of sup-interpretations: a survey. Theoretical Computer Science, Elsevier, 2012, pp.24. 10.1016/j.tcs.2012.11.003 . hal-00744915

HAL Id: hal-00744915

<https://hal.inria.fr/hal-00744915>

Submitted on 26 Oct 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Synthesis of sup-interpretations: a survey

Romain Péchoux

*Université de Lorraine and INRIA team Carte,
LORIA, Campus Scientifique - BP 239 - 54506 Vandoeuvre-lès-Nancy Cedex*

Abstract

In this paper, we survey the complexity of distinct methods that allow the programmer to synthesize a sup-interpretation, a function providing an upper-bound on the size of the output values computed by a program. It consists in a static space analysis tool without consideration of the time consumption. Although clearly related, sup-interpretation is independent from termination since it only provides an upper bound on the terminating computations. First, we study some undecidable properties of sup-interpretations from a theoretical point of view. Next, we fix term rewriting systems as our computational model and we show that a sup-interpretation can be obtained through the use of a well-known termination technique, the polynomial interpretations. The drawback is that such a method only applies to total functions (strongly normalizing programs). To overcome this problem we also study sup-interpretations through the notion of quasi-interpretation. Quasi-interpretations also suffer from a drawback that lies in the subterm property. This property drastically restricts the shape of the considered functions. Again we overcome this problem by introducing a new notion of interpretations mainly based on the dependency pairs method. We study the decidability and complexity of the sup-interpretation synthesis problem for all these three tools over sets of polynomials. Finally, we take benefit of some previous works on termination and runtime complexity to infer sup-interpretations.

Keywords: Complexity Analysis, Static Analysis, Resource Upper Bounds, Interpretation, Quasi-interpretation, Sup-interpretation

1. Introduction

1.1. Motivations

The notion of sup-interpretation was introduced in [1] in order to study program extensional complexity. This tool is devoted to statically analyze the complexity of programs guaranteeing that a secured system resists to buffer-overflows and thus allowing the programmer to verify complexity properties of

Email address: `Romain.Pechoux@loria.fr` (Romain Péchoux)

programs used in safety-critical systems. Sup-interpretations focus on analyzing the complexity of programs or, more specifically, term rewrite systems by considering upper bounds on the size of values computed by a program, by static analysis.

Basically, a sup-interpretation of a program is a function that provides an upper-bound on the size of the computed output with respect to the input size. In other words, given a program \mathbf{p} , the sup-interpretation of \mathbf{p} is a function that, given some input data x such that \mathbf{p} converges on input x , provides an upper-bound on the output size in the size of x .

One of the main issues concerning static analysis tools is related to their decidability and/or complexity. In other words, one tries to find if the static analysis is decidable and, if so, one tries to study its complexity. As highlighted by Rice's theorem, most of interesting (or non-trivial) analyses are undecidable and, in most of the cases, this issue is transformed into finding the complexity of a smaller instance of the initial problem. In the particular case of sup-interpretations, the analysis consists in finding the sup-interpretation of a given program, that is in synthesizing a function providing upper bounds on the program computations. We call this analysis the *sup-interpretation synthesis problem*. This paper will be dedicated to survey the results concerning the sup-interpretation (SI) synthesis problem.

1.2. Contribution.

The reader is assumed to be familiar with basic knowledge about term rewrite systems, see chapter 2 of [2] or [3], and computability and complexity, see [4, 5].

We start to show that the general problems of the sup-interpretation synthesis are undecidable when we consider functions and Gödel numberings. Moreover we show that the *sup-interpretation verification problem*, which consists in checking that a function given as input is a sup-interpretation, is Π_1^0 -complete in the arithmetical hierarchy and that the sup-interpretation synthesis problem is in Σ_3^0 .

Next we specify our language by introducing Term Rewriting Systems (TRS) and we define the corresponding notion of sup-interpretation. Starting from here, we will study well-known termination and complexity tools like polynomial interpretations (PI) and quasi-interpretations (QI) and show that they allow the programmer to obtain a sup-interpretation under some slight restrictions.

We demonstrate that (polynomial) interpretations for termination are special kind of sup-interpretations. However they were designed to study strong normalization and, consequently, they do not provide enough power to study programs computing partial recursive functions.

To overcome this problem, we study the notion of quasi-interpretation. We also show that quasi-interpretations define sup-interpretations.

Finally, we study a new notion called DP-interpretation (DPI) based on the

dependency pairs framework by Arts and Giesl [6] that also defines a sup-interpretation. We show that this new notion strictly generalizes the notion of quasi-interpretation since it does not require any subterm property, a property stating that considered interpretations have to be greater than each of their arguments. In other words, every program admitting a quasi-interpretation admits a DP-interpretation but the converse does not hold.

We study the sup-interpretation synthesis problem with respect to each of these tools on particular sets of polynomials ranging over a structure $\mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+, \mathbb{R}^+\}$. The considered sets of polynomials are:

- the set $\mathbb{K}[\overline{X}]$ of usual multivariate polynomials whose coefficients are in \mathbb{K} and with n variables $\overline{X} = X_1, \dots, X_n$ ranging over the field of real numbers,
- the set of $\text{MaxPoly}^{(k,d)}\{\mathbb{K}\}$ polynomials, which consist in functions obtained using constants over \mathbb{K} and arbitrary compositions of the operators $+, \times$ and \max of degree bounded by d and max arity bounded by k ,
- and the set of $\text{MaxPlus}^{(k,d)}\{\mathbb{K}\}$ functions, which consist in functions obtained using constants over \mathbb{K} bounded by d and arbitrary compositions of the operators $+$ and \max , with a max arity bounded by k .

The obtained results can be summarized by the following Figure:

Function space \ tool	PI	QI	DPI
$\mathbb{K}[\overline{X}], \mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+\}$	Undecidable	Undecidable	Undecidable
$\mathbb{R}^+[\overline{X}]$	Exptime	Exptime	Exptime
$\text{MaxPoly}\{\mathbb{K}\}, \mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+\}$	✘	Undecidable	Undecidable
$\text{MaxPoly}^{(k,d)}\{\mathbb{R}^+\}$	✘	Exptime	Exptime
$\text{MaxPlus}^{(k,d)}\{\mathbb{K}\}, \mathbb{K} \in \{\mathbb{N}, \mathbb{Q}_d^+\}$	✘	NP-complete	NP-complete
$\text{MaxPlus}^{(k,d)}\{\mathbb{R}^+\}$	✘	NP-hard	NP-hard

Figure 1: Decidability and complexity of the sup-interpretation synthesis problem

where \mathbb{Q}_d^+ consists in rationals of bounded representation.

The first line is direct consequences of Hilbert's tenth problem undecidability whereas the second line is a consequence of Tarski's quantifier elimination Theorem over real numbers. One important point to mention here is that the

synthesis problem is exponential and not doubly exponential because the synthesis problem is more restricted than general quantifier elimination.

In the first column, the symbol \boxtimes means that it does not make sense to study the synthesis problem with respect to the considered set of functions. Indeed the synthesis of polynomial interpretation has no meaning for any structure including a max operator since max is not a strictly monotonic function whereas polynomial interpretations deal with functions enjoying such a property.

The results for **MaxPoly** function space are identical to the results on pure polynomials since the max operator can be eliminated for both **QI** and **DPI**.

Finally, in the last two lines of Figure 1.2, we show that the synthesis problem is **NP-hard** for **MaxPlus**, independently of the structure. As a corollary, on bounded search spaces like \mathbb{N} or \mathbb{Q}_d^+ , the problem is **NP-complete**. The meaning of such a notion is unclear over an unbounded and uncountable space like \mathbb{R}^+ . Note that these results are a Corrigendum to results already presented in an unpublished workshop [7] that were wrongly stating a **NP-completeness** result over \mathbb{R}^+ .

Finally, we take benefit of termination results on the runtime complexity of TRS to infer sup-interpretations in a last section. In analogy with complexity theory, we show that time bounded computations imply size (or space) bounded computations. However the space bound may be exponential in the time, if the derivation length is the considered measure of time. Indeed, a derivation of length n may correspond to exponential space by just using variable duplication. We discuss the complexity of the synthesis problem for all of these termination techniques.

1.3. Outline

In Section 2 we consider general undecidable problems of the sup-interpretation synthesis when considering functions. In Section 3, we introduce Term Rewriting Systems and the corresponding notion of sup-interpretation that slightly differs from the sup-interpretation on functions. In Section 4, we introduce polynomial interpretations as sup-interpretations and study the decidability and complexity of their sup-interpretation synthesis problem. Sections 5 and 6 apply the same analysis to the notions of quasi-interpretation and DP-interpretation. Section 7 discusses the relation between time and space, where time is considered to be the derivation length and space is considered to be the size of a term. This section shows how to synthesize a sup-interpretation through the use of termination techniques. Finally, Section 8 discusses the main open issues.

1.4. Related works

Sup-interpretations are inspired by two former notions on Term Rewriting Systems, the polynomial interpretations, introduced in [8, 9] to analyze program termination and runtime complexity [10, 11, 12], and the quasi-interpretations, introduced in [13] and used to characterize complexity classes such as **FPTIME**, **FPSPACE** or **LOGSPACE** (See [14, 15, 16]).

The general framework of sup-interpretation was introduced in [1] without considering the synthesis problem. [17] was the first paper to combine interpretation methods together with the dependency pairs method in order to characterize polynomial time and space complexity classes in a more intensional way, that is by capturing more natural algorithms corresponding to a given polynomial time or space function. However the results were presented independently of the notion of sup-interpretation and the present paper gives a deeper understanding on the combination of both methods in order to obtain a sup-interpretation.

One important point to stress here is that sup-interpretations are an extensional tool contrarily to quasi-interpretations and polynomial interpretations that are intensional tools. It means that sup-interpretations deal with functions as mathematical object in the sense of complexity theory, that is functions computed by some programs, whereas (general) interpretations are intensional tools and deal with program properties. As a consequence, they also allow the programmer to study finer and more technical program behaviors. For example, a quasi-interpretation also provides upper-bounds on the size of a program intermediate computations whereas this property has no meaning for a sup-interpretation. However sup-interpretations can be combined in criteria in order to get intensional properties such as upper bounds on the size of intermediate values. The aim of this paper is neither to cover the way to get such intensional properties nor to show how they can help in characterizing complexity classes. Consequently, we encourage the interested reader to study [1].

The paper [18] has already deeply studied the synthesis problem for quasi-interpretations using max-polynomials with additive coefficient in \mathbb{N} or $\{0, 1\}$ and variables in \mathbb{Q}^+ . The present work takes advantage of these results to present them from a sup-interpretation point of view. Moreover they are extended, firstly, by considering rational and real multiplicative coefficients and, secondly, by extending the NP-hardness proof of [18] over \mathbb{N} to NP-completeness results over natural numbers and rational numbers of bounded representation (and not only $\{0, 1\}$). One last and important point is that the aim of the current paper is not to provide an automated way to synthesize a sup-interpretation but to find the complexity of the synthesis problem depending on the tool used (interpretation, quasi-interpretation, DP-interpretation, termination tools...), on the set of considered functions (polynomials, polynomials with max,...) and on the considered domain (positive real or rational numbers, natural numbers,...). The reader interested by automation should refer to the recent papers [19, 20, 21] that allow to build interpretations (and consequently, sup-interpretations as demonstrated in Section 4) for showing program termination and to the tools that synthesize quasi-interpretations [22, 23].

2. Undecidability results

In this section, we show undecidability results for the synthesis of sup-interpretations. All these results are machine independent and rely on simple Cantor's diagonalizations using Gödel numbering and s_n^m theorem:

Definition 1. Suppose that we have a fixed procedure that lists all the sequences of instructions. It associates the set of instructions P_x , the $(x + 1)$ st set of instructions in the list, to each integer x . x is called the Gödel number of P_x and it corresponds to the partial recursive function φ_x determined by P_x .

Theorem 1 (Kleene s_n^m [24]). $\forall m, n \geq 1$ there is a recursive function s_n^m of arity $m + 1$ such that $\forall x, y_1, \dots, y_m$:

$$\lambda z_1 \dots \lambda z_n \cdot \varphi_x(y_1, \dots, y_m, z_1, \dots, z_n) = \varphi_{s_n^m(x, y_1, \dots, y_m)}$$

In what follows, let PRF be the set of partial recursive functions and RF be the set of total recursive functions of domain and codomain \mathbb{N} . Given a function $f \in PRF$ and some number x , we write $f(x) \downarrow$ (respectively \downarrow_t) if f yields an output on input x (resp. in time t), and we write $f(x) \uparrow$ otherwise (resp. \uparrow_t otherwise). Consequently $f(x) \downarrow$ is equivalent to $\exists t, f(x) \downarrow_t$. μ is the classical minimization operator. Given a property $P(x)$, $\mu x.P(x)$ is the smallest x satisfying P .

Definition 2. Given a function $f \in PRF$, a sup-interpretation of f is a function $F \in RF$ that bounds f on its definition domain, i.e. $\forall x \in \mathbb{N}, f(x) \downarrow \implies F(x) \geq f(x)$.

First we can show as a direct consequence of Rice's Theorem that there exist partial recursive functions that do not have any recursive sup-interpretation:

Theorem 2.

$$\neg(\forall f \in PRF, \exists F \in RF, \forall x \in \mathbb{N}, f(x) \downarrow \implies F(x) \geq f(x))$$

Proof. Suppose that the implication $\forall f \in PRF, \exists F \in RF, \forall x \in \mathbb{N}, f(x) \downarrow \implies F(x) \geq f(x)$ holds. Define the function f by $f(n) = \varphi_n(n) + 1$. By definition, f is clearly in PRF . Consequently, $\exists F \in RF$ such that $\forall x \in \mathbb{N}, f(x) \downarrow \implies F(x) \geq f(x)$. Let i be the Gödel number of such a function F . We obtain that $\forall x \in \mathbb{N}, f(x) \downarrow \implies \forall x, \varphi_i(x) \geq f(x)$. As a consequence, $\varphi_i(i) \geq f(i) = \varphi_i(i) + 1$. It contradicts the hypothesis that $F \in RF$. \square

This diagonalization result no longer holds if we allow F to be in PRF (In this case, we can trivially set $F(x) = f(x)$).

Now we try to find a recursive function that given two Gödel numbers x and y would allow us to compare the corresponding partial recursive functions φ_x and φ_y . We also obtain a negative answer to this issue.

Theorem 3. $\nexists F \in RF$,

$$F(x, y) = \begin{cases} 1 & \text{if } \forall z, \varphi_x(z) \downarrow \implies \varphi_x(z) \leq \varphi_y(z) \\ 0 & \text{otherwise} \end{cases}$$

Proof. Suppose that such a recursive function F exists and define f to be the characteristic function of $\{ \langle x, y \rangle \mid \forall z, \varphi_x(z) \downarrow \implies \varphi_x(z) \leq \varphi_y(z) \}$. f

is recursive. Define ϕ to be a function of two variables corresponding to the following instructions set: given the input $\langle x, y \rangle$, apply P_x to x and return 0 if and when this computation converges. By Church-Turing thesis, it defines a partial recursive function:

$$\phi(x, y) = \begin{cases} 0 & \text{if } \varphi_x(x) \downarrow \\ \uparrow & \text{otherwise} \end{cases}$$

Suppose that i is the Gödel number of such a function, applying s_n^m Theorem, we obtain that there is a recursive function s_1^1 such that $\forall x, \lambda y. \phi(x, y) = \varphi_{s_1^1(i, x)}$. Now suppose that x_0 is a Gödel number for the constant function $\lambda x. 0$. We have that $\lambda x. f(s_1^1(i, x), x_0)$ is recursive since it is obtained by composition of recursive functions. However by definition:

$$\begin{aligned} f(s_1^1(i, x), x_0) &= \begin{cases} 1 & \text{if } \forall z, \varphi_{s_1^1(i, x)}(z) \downarrow \implies \varphi_{s_1^1(i, x)}(z) \leq 0 \\ 0 & \text{otherwise} \end{cases} \\ &= \begin{cases} 1 & \text{if } \varphi_{s_1^1(i, x)}(z) = 0 \\ 0 & \text{otherwise} \end{cases} \\ &= \begin{cases} 1 & \text{if } \varphi_x(x) \downarrow \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

So we have reduced our function to a variant of the halting problem (see Rogers [4]) which is known to be undecidable. Consequently, $\lambda x. f(s_1^1(i, x), x_0)$ is not recursive and we obtain a contradiction. \square

Consequently, we obtain that the *sup-interpretation verification problem* defined by $SI(F) = \{x \mid \forall z \varphi_x(z) \downarrow \implies \varphi_x(z) \leq F(z)\}$, which consists in checking that a given function F is a sup-interpretation of a function f of index x (i.e. $x \in SI(F)$), is undecidable. As a corollary, we also obtain that the *sup-interpretation synthesis problem*, which consists in finding the smallest function wrt Gödel numbering that bounds another given as input, is also undecidable:

Corollary 1. $\exists G \in RF$ such that:

$$G(x) = \begin{cases} \mu y. \{\forall z, \varphi_x(z) \downarrow \implies \varphi_x(z) \leq \varphi_y(z)\} \\ 0 & \text{otherwise} \end{cases}$$

Proof. Assume that G is recursive and that we have a Gödel numbering starting from Gödel number 1 (i.e. not defined in 0). The reason for which we take such a numbering is just that we do not want to make a confusion between the output 0 when there is no upper-bound and the index 0 of the function φ_0 that might be an upper bound of some other function. Then $\mu y. F(x, y)$, with F defined in Theorem 3 has the same characteristic function than G . Consequently, we obtain a contradiction and G cannot be recursive. \square

Now let just state that the sup-interpretation verification problem which consists in checking that a fixed function F is a sup-interpretation of a function φ_x of index x , noted $SI(F)$ is Π_1^0 -complete in the arithmetical hierarchy:

Theorem 4. *The sup-interpretation verification problem $SI(F)$ is Π_1^0 -complete.*

Proof. For every input z and every t , either $\varphi_x(z)$ terminates within time t and, in this case, we have to compare φ_x and F or $\varphi_x(z)$ does not terminate in time t . Consequently, we can write $SI(F) = \{x \mid \forall z, \forall t, \varphi_x(z) \uparrow_t \vee (\varphi_x(z) \downarrow_t \wedge \varphi_x(z) \leq F(z))\}$.

We briefly recall that a problem B is complete for some class C of the arithmetical hierarchy if there is a total computable function f such that:

$$x \in A \text{ iff } f(x) \in B$$

for some problem A known to be C -complete. Consider the problem $A = \{x \mid \varphi_x(0) \uparrow\}$. This problem is known to be Π_1^0 -complete since it is co-RE. Now define the function f such that for each x the function $\varphi_{f(x)}$ of index $f(x)$ is defined by:

$$\varphi_{f(x)} = \begin{cases} F(z) + 1 & \text{if } \varphi_x(0) \downarrow \\ \uparrow & \text{otherwise} \end{cases}$$

We clearly have:

$$x \in A \text{ iff } f(x) \in SI(F)$$

Moreover the function f is clearly total, by definition, and computable, by applying s_n^m Theorem. Consequently, $SI(F)$ is Π_1^0 -complete. \square

Now we show that sup-interpretation synthesis problem, SI defined to be “the set of functions $f \in PRF$ for which there is a total recursive function F , satisfying: for all $z \in \mathbb{N}$ if $f(z) \downarrow$ then $F(z) \geq f(z)$ ” is Σ_3^0 in the arithmetical hierarchy:

Theorem 5. $SI \in \Sigma_3^0$.

Proof. SI can be written equivalently as:

$$SI = \{x \in \mathbb{N} \mid \exists s, \forall z, \exists t, \varphi_s(z) \downarrow_t \wedge (\varphi_x(z) \downarrow_t \implies \varphi_s(z) \geq \varphi_x(z))\}$$

In other words, SI is the the set of indexes x corresponding to functions φ_x for which there exists a total function φ_s providing an upper bound on terminating computations (Indeed $\varphi_x(z) \downarrow_t \implies \varphi_s(z) \geq \varphi_x(z)$). The formula $\varphi_s(z) \downarrow_t \wedge (\varphi_x(z) \downarrow_t \implies \varphi_s(z) \geq \varphi_x(z)) \in \Pi_0^0$ and, consequently, $SI \in \Sigma_3^0$. \square

3. Sup-interpretations over Term Rewriting Systems

3.1. TRS as a computational model

The previous section only deals with machine independent results and we have hidden for a while the data representation problems arising. Consequently, we have to adapt slightly the notion of sup-interpretation to each computational model under consideration. Throughout the following Sections, we will consider term rewriting systems.

A Term Rewriting System (TRS for short) is a formal system for manipulating terms over a signature by means of rules.

Terms are strings of symbols consisting of a countably infinite set of variables Var and a first-order signature Σ , a non-empty set of function symbols or operator symbols of fixed arity. Var and Σ are supposed to be disjoint. As usual, the notation $Ter(\Sigma, Var)$ will be used to denote the set of terms s, t, \dots of signature Σ and having variables in Var .

A (one-hole) context $C[\diamond]$ is a term in $Ter(\Sigma \cup \{\diamond\}, Var)$ with exactly one occurrence of the hole \diamond , a symbol of arity 0. Given a term t and context $C[\diamond]$, let $C[t]$ denote the result of replacing the hole \diamond with the term t .

A substitution σ is a mapping from Var to $Ter(\Sigma, Var)$.

A rewrite rule for a signature Σ is a pair $l \rightarrow r$ of terms $l, r \in Ter(\Sigma, Var)$. A Term Rewrite System is as a pair $\langle \Sigma, \mathcal{R} \rangle$ of a signature Σ and a set of rewrite rules \mathcal{R} . In what follows, we will suppose that all the variables of a right-hand side r are included in the variables of l as in Chapter 2 of [2].

A constructor Term Rewrite System is a TRS in which the signature Σ can be partitioned into the disjoint union of a set of function symbols \mathcal{D} and a set of constructors \mathcal{C} , such that for every rewrite rule $l \rightarrow r$ we have $l = \mathbf{f}(t_1, \dots, t_n)$ with $\mathbf{f} \in \mathcal{D}$ and $t_1, \dots, t_n \in Ter(\mathcal{C}, Var)$. The constructors are introduced to represent inductive data. They basically consist of a strict subset $\mathcal{C} \subset \Sigma$ of non-defined functions (a function is defined if it is the root of a left-hand side term in a rule). In what follows, we will only consider constructor TRS and we will use the notation $\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle$ to denote such a particular TRS, $\mathcal{C} \uplus \mathcal{D}$ being the disjoint union of the sets \mathcal{C} and \mathcal{D} . Terms in $Ter(\mathcal{C}, Var)$ will be called patterns. In what follows, we will consider orthogonal constructor TRS since we only want to deal with functions. The notion of orthogonality requires that reduction rules of the system are all left-linear, that is each variable occurs only once on the left hand side of each rule, and there is no overlap between patterns. It is a sufficient condition to ensure that the considered TRS is confluent. It implies that we are clearly talking of functions that maps a term to another (and not functions mapping a term to a set of terms in the case of non-confluent systems). Note that this syntactic requirement could have been withdrawn in favor of a semantic restriction that would only consider TRS that compute functions. Our choice restricts the expressivity of considered TRS but makes sense in our theoretical development since it does not restrict the computed functions set.

Given two terms s and t , we have that $s \rightarrow_{\mathcal{R}} t$ if there are a substitution σ , a context $C[\diamond]$ and a rule $l \rightarrow r \in \mathcal{R}$ such that $s = C[l\sigma]$ and $t = C[r\sigma]$. Throughout the paper, let $\rightarrow_{\mathcal{R}}^*$ (resp. $\rightarrow_{\mathcal{R}}^+$) be the reflexive and transitive (resp. transitive) closure of $\rightarrow_{\mathcal{R}}$. Moreover we write $s \rightarrow_{\mathcal{R}}^n t$ if n rewrite steps are performed to rewrite s to t . A TRS terminates if there is no infinite reduction through $\rightarrow_{\mathcal{R}}$. A function symbol \mathbf{f} of arity n will define a partial function $\llbracket \mathbf{f} \rrbracket$ from constructor

terms¹ (sometimes called values) $Ter(\mathcal{C})^n$ to $Ter(\mathcal{C})$ by:

$$\forall v_1, \dots, v_n \in Ter(\mathcal{C}), \llbracket \mathbf{f} \rrbracket(v_1, \dots, v_n) = v \text{ iff } \mathbf{f}(v_1, \dots, v_n) \rightarrow_{\mathcal{R}}^* v \wedge v \in Ter(\mathcal{C})$$

In this case, we write $\llbracket \mathbf{f} \rrbracket(v_1, \dots, v_n) \downarrow$ to mean that the computation ends in a normal form (constructor term). If there is no such a v (because of divergence or because evaluation cannot reach a constructor term), then $\llbracket \mathbf{f} \rrbracket(v_1, \dots, v_n) \uparrow$. Finally, we define the notion of size of a term $|e|$ which is equal to the number of symbols in e .

3.2. Sup-interpretation of a TRS

Since the goal of sup-interpretation is to provide a non-negative upper bound on the size of computed values, we will mainly restrict our analysis to the groups \mathbb{N}, \mathbb{Q}^+ and \mathbb{R}^+ , where \mathbb{Q}^+ and \mathbb{R}^+ denote positive rational numbers and positive real numbers. In what follows, let $\mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+, \mathbb{R}^+\}$ and let \geq and $>$ be the natural ordering and strict ordering on such a structure. Finally, let $>_\delta$ be the strict ordering defined by $x >_\delta y$ iff $x \geq \delta + y$, for some fixed $\delta \in \mathbb{K}$ such that $\delta > 0$.

Definition 3. *Given a TRS $\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle$, an assignment θ over \mathbb{K} is a mapping that maps every symbol $\mathbf{g} \in \mathcal{D} \uplus \mathcal{C}$ of arity m to a total function $\theta(\mathbf{g}) : \mathbb{K}^m \rightarrow \mathbb{K}$ and that maps every variable $\in Var$ to a variable in \mathbb{K} .*

An assignment is additive if $\forall c \in \mathcal{C}$ of arity $n > 0$, $\theta(c) = \lambda x_1, \dots, x_n. (x_1 + \dots + x_n + k_c)$, for some $k_c \geq 1$, and $\forall c \in \mathcal{C}$ of arity 0, $\theta(c) = 0$. An assignment is k -additive if for all $c \in \mathcal{C}$, $k_c \leq k$.

Definition 4. *An assignment θ over \mathbb{K} is (strictly) monotonic if for every symbol \mathbf{f} of arity m , $\theta(\mathbf{f})$ is a (strictly) monotonic function in each of its arguments. In other words, $\forall i \in [1, m], x \geq y \implies \theta(\mathbf{f})(\dots, x_{i-1}, x, x_{i+1}, \dots) \geq \theta(\mathbf{f})(\dots, x_{i-1}, y, x_{i+1}, \dots)$ (resp. $\forall i \in [1, m], \forall \delta > 0, \exists \epsilon > 0, \theta(\mathbf{f})(\dots, x + \delta, \dots) >_\epsilon \theta(\mathbf{f})(\dots, x, \dots)$).*

Now we are able to adapt the notion of sup-interpretation to this model:

Definition 5 (Sup-interpretation). *Given a TRS $\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle$, a monotonic and additive assignment θ over \mathbb{K} is a sup-interpretation over \mathbb{K} if $\forall \mathbf{f} \in \mathcal{D}$ of arity m and $\forall v_1, \dots, v_m \in Ter(\mathcal{C})$:*

$$\mathbf{f}(v_1, \dots, v_m) \downarrow \implies \theta(\mathbf{f}(v_1, \dots, v_m)) \geq \theta(\llbracket \mathbf{f} \rrbracket(v_1, \dots, v_m))$$

where the sup-interpretation θ is extended canonically to general terms by:

$$\theta(\mathbf{g}(\mathbf{e}_1, \dots, \mathbf{e}_n)) = \theta(\mathbf{g})(\theta(\mathbf{e}_1), \dots, \theta(\mathbf{e}_n)), \quad \mathbf{g} \in \mathcal{D} \uplus \mathcal{C}$$

¹As usual $Ter(\mathcal{C}) = Ter(\mathcal{C}, \emptyset)$

We restrict the shape of constructor symbol sup-interpretations by requiring a k -additive assignment. This restriction is made to relate easily the interpretation of a constructor term and its size, i.e. $\exists k \in \mathbb{N}, \forall v \in \text{Ter}(\mathcal{C}), k \times |v| \geq \theta(v) \geq |v|$ always hold for a TRS wrt a fixed additive sup-interpretation.

We compare this new definition wrt the one presented in previous Section: in a given TRS, the sup-interpretation of a function symbol \mathbf{f} of arity m can be discretized to be viewed as a function $\theta(\mathbf{f}) : \mathbb{N}^m \rightarrow \mathbb{N}$ that bounds the size of the output wrt to the input sizes (this is direct for 1-additive sup-interpretations):

Lemma 1. *Given a TRS $\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle$ having a sup-interpretation θ then for each function symbol $\mathbf{f} \in \mathcal{D}$ and for all values $v_1, \dots, v_m \in \text{Ter}(\mathcal{C})$ such that $\mathbf{f}(v_1, \dots, v_m) \downarrow$, we have:*

$$\theta(\mathbf{f})(k \times |v_1|, \dots, k \times |v_m|) \geq |[\mathbf{f}](v_1, \dots, v_m)|$$

Proof.

$$\begin{aligned} \theta(\mathbf{f})(k \times |v_1|, \dots, k \times |v_m|) & \quad \text{By monotonicity} \\ & \geq \theta(\mathbf{f})(\theta(v_1), \dots, \theta(v_m)) & \quad \text{and } k\text{-additivity} \\ & = \theta(\mathbf{f}(v_1, \dots, v_m)) & \quad \text{By extension} \\ & \geq \theta([\mathbf{f}](v_1, \dots, v_m)) & \quad \text{By Definition 5} \\ & \geq |[\mathbf{f}](v_1, \dots, v_m)| & \quad \text{By } k\text{-additivity} \end{aligned}$$

and so the conclusion. \square

4. Polynomial interpretations

4.1. Interpretations as sup-interpretations

Given a TRS, the main issue is now to synthesize a sup-interpretation, that is to compute an upper-bound on the partial function it computes. The first natural technique to do so comes from the term rewriting termination community, is called (polynomial) interpretation and was introduced in [9, 8].

Definition 6 (Interpretation). *Given a TRS $\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle$, an (additive) interpretation is a strictly monotonic (additive) assignment $[-]$ over \mathbb{K} which satisfies:*

1. $\forall l \rightarrow r \in \mathcal{R}, [l] > [r]$
2. If $\mathbb{K} \in \{\mathbb{Q}^+, \mathbb{R}^+\}$ then:
 - (a) either $\forall \mathbf{g} \in \mathcal{D} \uplus \mathcal{C}$, of arity $m > 0$,
 $\forall i \in [1, m], [\mathbf{g}](X_1, \dots, X_m) > X_i$
 - (b) or $\forall l \rightarrow r \in \mathcal{R}, [l] >_\delta [r]$

where the interpretation $[-]$ is extended canonically to terms as usual.

Condition 1 constitutes the basis of interpretation method as introduced in [9, 8]. Condition 2(a) was introduced by Dershowitz [25] to compensate for the loss of well-foundedness over the reals. Finally, condition 2(b) is due to Lucas [26] and captures more TRS than 2(a).

As demonstrated in [8], an interpretation defines a reduction ordering (i.e. a strict, stable, monotonic and well-founded ordering)

Theorem 6. *If a TRS $\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle$ admits an interpretation then it terminates.*

Moreover, an additive interpretation defines a sup-interpretation:

Theorem 7. *Given a TRS $\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle$ having an additive interpretation $[-]$ then $[-]$ is a sup-interpretation.*

Proof. First note that the assignment is additive by assumption. Second, we show that for each values $v_1, \dots, v_n \in \text{Ter}(\mathcal{C})$ and function symbol $\mathbf{f} \in \mathcal{D}$ such that $\mathbf{f}(v_1, \dots, v_n) \downarrow$ we have $[\mathbf{f}(v_1, \dots, v_n)] \geq [[\mathbf{f}]](v_1, \dots, v_n)$. Consider a function symbol \mathbf{f} and values v_1, \dots, v_n , by Theorem 6, we have $\mathbf{f}(v_1, \dots, v_n) \downarrow$. Since interpretations define a reduction ordering, we have that each reduction corresponds to a (strictly) decreasing sequence:

$$\begin{aligned} \mathbf{f}(v_1, \dots, v_n) &\rightarrow_{\mathcal{R}} u_1 \rightarrow_{\mathcal{R}} \dots \rightarrow_{\mathcal{R}} u_k \rightarrow_{\mathcal{R}} [[\mathbf{f}]](v_1, \dots, v_n) \\ [\mathbf{f}(v_1, \dots, v_n)] &> [u_1] > \dots > [u_k] > [[\mathbf{f}]](v_1, \dots, v_n) \end{aligned}$$

and, a fortiori, $[\mathbf{f}(v_1, \dots, v_n)] \geq [[\mathbf{f}]](v_1, \dots, v_n)$. □

Consequently, finding the interpretation of a given program provides a sup-interpretation of this program under additivity constraints as illustrated by the following example:

Example 1. *Consider the following simple TRS:*

$$\begin{array}{ll} \mathbf{d}(0) \rightarrow 0 & \mathbf{exp}(0) \rightarrow 1 \\ \mathbf{d}(x+1) \rightarrow \mathbf{d}(x) + 2 & \mathbf{exp}(x+1) \rightarrow \mathbf{d}(\mathbf{exp}(x)) \end{array}$$

where $x+2$ and 1 are notations for $(x+1)+1$ and $0+1$. It admits the following additive interpretation $[0] = 0$, $[+1](X) = X+1$, $[\mathbf{d}](X) = 3 \times X + 1$, $[\mathbf{exp}](X) = 3^{2 \times X + 1}$. Indeed, it is a strictly monotonic additive assignment and for the last rule, we have:

$$\begin{aligned} [\mathbf{exp}](x+1) &= 3^{2 \times [(x+1)+1]} = 3^{2(X+1)+1} = 3^{2X+3} \\ &> 3 \times 3^{2X+1} + 1 = [\mathbf{d}](3^{2X+1}) = [\mathbf{d}](\mathbf{exp}(x)) \end{aligned}$$

We let the reader check that the strict inequalities hold for the other rules.

4.2. Restriction to polynomials

It is natural to restrict the space of considered functions (the sup-interpretation codomain) to polynomials for two reasons. First, as we have seen in the first Section, considering the whole space of functions is too general in terms of decidability. Second, polynomials are admitted to be a relevant set of functions in term of time and space complexity. Consequently, we restrict the function space in order to get effective procedures.

In what follows, let $\mathbb{K}[X_1, \dots, X_m]$ be the set of m -ary polynomials whose coefficients are in \mathbb{K} .

Definition 7 (Polynomial interpretation). *Given a TRS $\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle$, a polynomial interpretation over \mathbb{K} is an interpretation $[-]$ over \mathbb{K} that maps every symbol $g \in \mathcal{D} \uplus \mathcal{C}$ of arity m to a function $[g] \in \mathbb{K}[X_1, \dots, X_m]$.*

The synthesis problem for polynomial interpretation has been deeply studied in [27, 28] where algorithms solving the constraints are described. More recently, encoding-based algorithms via SAT or SMT solving have become the state of the art for the synthesis problem [19, 21]. One important question is what is the best structure (\mathbb{N}, \mathbb{Q}^+ or \mathbb{R}^+) to consider in order to get a polynomial interpretation. This question has no answer as surveyed by the following results:

Theorem 8 (Lucas [26]). *There are TRS that can be proved terminating using a polynomial interpretation over \mathbb{R} , whereas they cannot be proved terminating using a polynomial interpretation over \mathbb{Q} .*

Theorem 9 (Lucas [26]). *There are TRS which can be proved terminating using a polynomial interpretation over \mathbb{Q} , whereas they cannot be proved terminating using a polynomial interpretation over \mathbb{N} .*

Theorem 10 (Middeldorp-Neurauter. [29]). *There are TRS which can be proved terminating using a polynomial interpretation over \mathbb{N} , whereas they cannot be proved terminating using a polynomial interpretation over \mathbb{Q} or \mathbb{R} .*

4.3. Decidability results over polynomials

However we can compare the structures through decidability or undecidability results for the sup-interpretation synthesis problem as illustrated below.

Definition 8 (PI synthesis problem). *Given a TRS $\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle$, is there an assignment $[-]$ such that $[-]$ is a polynomial interpretation of $\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle$?*

Theorem 11. *The PI synthesis problem is undecidable over $\mathbb{N}[\overline{X}]$ and $\mathbb{Q}^+[\overline{X}]$.*

Proof. This is a direct consequence of Hilbert’s tenth Problem undecidability since every inequality of Definition 7 of the shape $\forall[\mathbf{x}_1], \dots, [\mathbf{x}_n], [l] > [r]$, $\mathbf{x}_1, \dots, \mathbf{x}_n$ being the free variables of l , can be turned into the satisfaction of the formula $\neg\exists[\mathbf{x}_1], \dots, [\mathbf{x}_n], [l] - [r] = 0$. The interested reader should refer to [30]. Note that we have not checked that each arbitrary polynomial can be encoded. This technical check which is needed to show a reduction from Hilbert’s tenth problem to the PI synthesis will be performed in the next section for the notion of quasi-interpretation. \square

This result was historically mentioned to be undecidable by Lankford [8].

Now we show that the polynomial interpretation synthesis problem is decidable over \mathbb{R}^+ as a corollary of Tarski’s Theorem [31]. Historically, Tarski’s procedure was non-elementary. It has been improved by Collins [32] in a procedure of complexity doubly exponential in the number of variables. We will use the most precise upper bound on such a procedure known by the author

and described in [33], where the procedure is shown to be doubly exponential in the number of quantifiers blocks alternations and exponential in the number of variables, in order to exhibit a precise upper bound on the complexity of the PI synthesis problem: we will obtain an *exponential* procedure because the polynomial quasi-interpretation synthesis problem is more restricted than the general quantifiers elimination over \mathbb{R}^+ described by Tarski.

Theorem 12 (Roy et Al. [33]). *Given an integral domain \mathbf{k} (i.e. a commutative ring with no zero divisor) included in a real closed field \mathbf{R} , a formula ϕ of size L in the ordered fields language under prenex normal form with parameters in \mathbf{K} , containing m blocks of quantifiers and s polynomials of n variables and with coefficients in \mathbf{k} whose sum of degrees is less or equal to D , there is an algorithm of complexity $O(L)D^{n^{O(m)}}$ which computes an equivalent quantifier-free formula.*

Theorem 13. *The PI synthesis problem is decidable in exponential time (in the size of the program) over $\mathbb{R}^+[\overline{X}]$.*

Proof. We start by encoding the strict monotonicity property: Given a TRS $\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle$, $\mathbf{f} \in \mathcal{D}$ of arity n and an assignment $[-] \in \mathbb{R}^+[\overline{X}]$ such that $[\mathbf{f}]$ is defined, the strict monotonicity property can be encoded by the following first order formula:

$$SM[\mathbf{f}] = \forall X_1, \dots, X_n, \forall Y_1, \dots, Y_n, \\ \bigwedge_{l \in [1, n]} X_l > Y_l \implies [\mathbf{f}](X_1, \dots, X_n) > [\mathbf{f}](Y_1, \dots, Y_n)$$

In other words, $SM[\mathbf{f}]$ if and only if $[\mathbf{f}]$ is strictly monotonic.

Now we encode the inequalities for each rule of a given program $\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle$: Given a TRS $\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle$, of assignment $[-]$, let \overline{a} be an enumeration of the multiplicative coefficients involved in the polynomials $[\mathbf{f}]$, $\forall \mathbf{f} \in \mathcal{D} \uplus \mathcal{C}$, and define $PI[\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle] = \exists \overline{a} \in \mathbb{R}^+, (\bigwedge_{\mathbf{f} \in \mathcal{D} \uplus \mathcal{C}} SM[\mathbf{f}]) \wedge (\bigwedge_{l \rightarrow r \in \mathcal{R}} [l] > [r])$.

$PI[\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle]$ is true if and only if there is an assignment $[-]$ that is a polynomial interpretation of $\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle$.

Performing a careful α -conversion of all the variables occurring in the distinct inequalities of the formula $PI[\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle]$, we can extrude all the quantifiers (existential and universal) to obtain a new formula under prenex normal form with only one alternation between a block of existential quantifiers (encoding the polynomials multiplicative coefficients) and one block of universal quantifiers (encoding program variables).

Now we apply Theorem 12 by setting $\mathbf{K} = \mathbb{R}$ and $\phi = PI[\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle]$ and we obtain an algorithm of complexity $O(|PI[\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle]|)D^{n^{O(m)}}$ which computes an equivalent quantifier-free formula. Note that:

- the size of the formula $PI[\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle]$ is bounded polynomially by the size of the program and exponentially by the maximal degree of the polynomial, which is also bounded by D . Indeed the number of multiplicative coefficients within a polynomial of bounded degree D is exponential in D .

- the number n of variables is bounded polynomially by the size of the program and exponentially by the degree D
- the number m of blocks is bounded by 2

Consequently, the algorithm has a complexity exponential in the size of the program. \square

4.4. Drawbacks of (polynomial) interpretations

The previous Subsection has provided a positive result, that is a mechanical way to synthesize the sup-interpretation of a given program. On the other hand, Theorem 6 can be interpreted as a negative result. Indeed, in terms of TRS, termination means that either the evaluation stops on a constructor term $v \in \text{Ter}(\mathcal{C})$ or that the evaluation stops on a (undefined) term still containing non-evaluated function symbols in \mathcal{D} . In particular, it means that this analysis rejects all the partial functions that diverge on some input domain but still remain bounded on its complement, as illustrated by the following example:

Example 2.

$$\mathbf{f}(x + 2) \rightarrow \mathbf{f}(x) + 2 \qquad \mathbf{f}(0) \rightarrow \mathbf{f}(0) \qquad \mathbf{f}(1) \rightarrow 1$$

The function \mathbf{f} computes the identity function on odd numbers whereas it infinitely diverges on even numbers. Consequently, it does not admit any polynomial interpretation whereas we would expect $\theta(\mathbf{f})(X) = X$ to be a suitable sup-interpretation.

5. Quasi-interpretations

5.1. Quasi-interpretations as sup-interpretations

We introduce the notion of quasi-interpretation [34] that, in contrast with (polynomial) interpretations, allows us to study partial functions.

Definition 9 (Quasi-interpretation). *Given a TRS $\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle$, a (additive) quasi-interpretation (QI for short) is a monotonic (additive) assignment $\llbracket - \rrbracket$ over \mathbb{K} satisfying:*

1. $\forall l \rightarrow r \in \mathcal{R}, \llbracket l \rrbracket \geq \llbracket r \rrbracket$
2. $\forall \mathbf{g} \in \mathcal{D} \uplus \mathcal{C}$, of arity m , $\forall i \in [1, m], \llbracket \mathbf{g} \rrbracket(X_1, \dots, X_m) \geq X_i$

where the quasi-interpretation $\llbracket - \rrbracket$ is extended canonically to terms as usual.

Condition 2 is called the *subterm property*. Quasi-interpretations do not tell anything about program termination since the strict ordering of Definition 6 has been replaced by its reflexive closure. Well-foundedness is lost and this is the main reason why such a tool can be adapted to partial functions. With this notion, we obtain a result similar to Theorem 7.

Theorem 14. *Given a program $\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle$ having an additive quasi-interpretation $\langle \!|-\!| \rangle$ then $\langle \!|-\!| \rangle$ is a sup-interpretation.*

Proof. The proof is essentially the same as the one in Theorem 7. Strict inequalities are replaced by non-strict inequalities. \square

Example 3. *The program of Example 2 admits the following additive quasi-interpretation: $\langle \!|0\!| \rangle = 0$, $\langle \!|+1\!| \rangle(X) = X + 1$ and $\langle \!|f\!| \rangle(X) = X$. Indeed, for the first rule, we check:*

$$\begin{aligned} \langle \!|f(x+2)\!| \rangle &= \langle \!|f\!| \rangle(\langle \!|(x+1)+1\!| \rangle) = X + 2 \\ &\geq \langle \!|f(x)\!| \rangle + 2 = \langle \!|(f(x)+1)+1\!| \rangle \end{aligned}$$

For the second, rule we clearly have $\langle \!|f(0)\!| \rangle \geq \langle \!|f(0)\!| \rangle$ and, for the last rule, we have $\langle \!|f(1)\!| \rangle = \langle \!|f\!| \rangle(\langle \!|1\!| \rangle) \geq \langle \!|1\!| \rangle$.

5.2. Quasi-interpretation synthesis problem

The quasi-interpretation synthesis problem was introduced by Amadio in [18] and is prominent in the perspective of practical uses of quasi-interpretation since an algorithm synthesizing a quasi-interpretation of a given program would allow the programmer to automatically perform a static analysis of program resources use on terminating computations. It can be defined as follows:

Definition 10 (QI synthesis problem). *Given a TRS $\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle$, is there an assignment $\langle \!|-\!| \rangle$ such that $\langle \!|-\!| \rangle$ is a quasi-interpretation of $\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle$?*

This problem is undecidable in the general case where we consider total functions as a consequence of Rice's Theorem and as illustrated by Corollary 1. Indeed there is no function (and consequently no program) that for a program index given as input provides the smallest index of a sup-interpretation. Consequently, we have to restrict again the set of considered functions. The immediate candidate is the set of polynomials presented in the previous Section. However we choose to add an extra max function. There are many reasons to do so: firstly, max is the smallest function satisfying the subterm condition. Thus it provides the tightest upper bound that we could expect on a function symbol computation. Secondly, it remains stable for the set of polynomials since the max is always bounded by the sum. Lastly, it was not considered in polynomial interpretations for the only reason that it is not strictly monotonic in each of its arguments (i.e. $x > x' \Rightarrow \max(x, y) > \max(x', y)$ does not hold in the case where $y > x$ with $x, x', y \in \mathbb{K}$). We define the set of MaxPoly functions as follows:

Definition 11. *Let $\text{MaxPoly} \{ \mathbb{K} \}$ be the set of functions obtained using constants and variables ranging over \mathbb{K} and arbitrary compositions of the operators $+$, \times and \max .*

We exhibit a normalization result on such a set of functions showing that max operator can be restricted to the upper most level:

Proposition 1 (Normalisation). *Each function $Q \in \text{MaxPoly}\{\mathbb{K}\}$, $Q \neq 0$, can be written into the following normal form:*

$$Q(X_1, \dots, X_n) = \max(P_1(X_1, \dots, X_n), \dots, P_k(X_1, \dots, X_n))$$

for some $k \geq 1$ and where $P_i \neq 0$ are polynomials.

Proof. By induction on the structure of Q :

- The base case is when Q is a monomial then $Q = \max(Q)$.
- If $Q = Q_1 + Q_2$ then by induction hypothesis $Q_i = \max(P_1^i, \dots, P_{n_i}^i)$, for $i \in \{1, 2\}$, with P_j^i polynomials. Consequently, $Q = \max(P_1^1, \dots, P_{n_1}^1) + \max(P_1^2, \dots, P_{n_2}^2) = \max_{j \leq n_1, k \leq n_2} (P_j^1 + P_k^2)$ since the max operator can be extruded using rules of the shape $\max(Q, R) + P = \max(Q + P, R + P)$ and $\max(\max(P, Q), \max(R, S)) = \max(P, Q, R, S)$.
- In the same way, if $Q = Q_1 \times Q_2$ then $Q = \max_{j \leq n_1, k \leq n_2} (P_j^1 \times P_k^2)$

and so the conclusion. \square

Moreover, we show that the satisfaction of an inequality in $\text{MaxPoly}\{\mathbb{K}\}$ can be transformed into an equivalent problem over polynomials, that is an inequality over $\text{MaxPoly}\{\mathbb{K}\}$ can be turned into a conjunction of disjunctions of inequalities over polynomials:

Proposition 2. *Given an inequality $Q \geq Q'$, with $Q, Q' \in \text{MaxPoly}\{\mathbb{K}\}$ there are two integers n and m and polynomials over \mathbb{K} , P_i, R_j for $i \leq n$, $j \leq m$, such that:*

$$Q \geq Q' \text{ iff } \bigwedge_{j \in [1, m]} \bigvee_{i \in [1, n]} P_i \geq R_j$$

Proof. By the previous Proposition, Q and Q' can be written as $\max(P_1, \dots, P_n)$ and $\max(R_1, \dots, R_m)$, for some n and m . Consequently:

$$\begin{aligned} \max(P_1, \dots, P_n) &\geq \max(R_1, \dots, R_m) \\ \Leftrightarrow \bigwedge_{j \in [1, m]} \max(P_1, \dots, P_n) &\geq R_j \\ \Leftrightarrow \bigwedge_{j \in [1, m]} \bigvee_{i \in [1, n]} P_i &\geq R_j \end{aligned}$$

and so the result holds. \square

5.3. Undecidable synthesis over Max-Poly $\{\mathbb{N}\}$

As expected, the QI synthesis problem remains undecidable over \mathbb{N} and \mathbb{Q}^+ :

Theorem 15. *The QI synthesis problem is undecidable over $\text{MaxPoly}\{\mathbb{N}\}$ and $\text{MaxPoly}\{\mathbb{Q}^+\}$.*

Proof. We demonstrate, using Proposition 2, that the synthesis problem over $\text{MaxPoly}\{\mathbb{N}\}$ and $\text{MaxPoly}\{\mathbb{Q}^+\}$ can be turned in the satisfaction of (disjunctions and conjunctions of) inequalities of the shape²:

$$\exists a_1, \dots, a_n \forall x_1, \dots, x_m, P(a_1, \dots, a_n, x_1, \dots, x_m) \geq 0$$

where the a_i represent the multiplicative coefficients of the function symbols quasi-interpretations and where the x_j represent the program variables quasi-interpretations. Fixing the a_i , this problem consists in checking that:

$$\forall x_1, \dots, x_m, P'(x_1, \dots, x_m) \geq 0$$

with $P'(x_1, \dots, x_m) = P(a_1, \dots, a_n, x_1, \dots, x_m)$. Now we consider Hilbert's tenth problem that was shown to be undecidable over \mathbb{Q}^+ (and \mathbb{N}) by Matijasevich [35]. Given a polynomial P of arity n , there is no procedure that decides:

$$\exists x_1, \dots, x_n, P(x_1, \dots, x_n) = 0$$

Over \mathbb{N} , we have:

$$\begin{aligned} & \exists x_1, \dots, x_n, P(x_1, \dots, x_n) = 0 \\ & \iff \neg(\forall x_1, \dots, x_n, P(x_1, \dots, x_n)^2 > 0) \\ & \iff \neg(\forall x_1, \dots, x_n, P(x_1, \dots, x_n)^2 - 1 \geq 0) \end{aligned}$$

Given a polynomial P , having a computable procedure that checks whether $\forall x_1, \dots, x_n, P(x_1, \dots, x_n)^2 - 1 \geq 0$ holds would provide a positive answer to Hilbert's problem (and conversely). As a consequence, we know that there is no such a procedure. Finally, we check (a technical but not difficult fact) that for any polynomial P of arity n we can enforce the interpretation of a n -ary symbol \mathbf{f} to satisfy $\llbracket \mathbf{f} \rrbracket(x_1, \dots, x_n) = P(x_1, \dots, x_n)^2$ and $\forall x_1, \dots, x_n, P(x_1, \dots, x_n)^2 \geq 1$ adding arbitrary rules to a program (provided that $\llbracket \mathbf{f} \rrbracket \in \text{MaxPoly}\{\mathbb{N}\}$). For simplicity, suppose that we have additive constructors \mathbf{c}_n of arity $n \in \mathbb{N}$ and such that $\llbracket \mathbf{c}_0 \rrbracket = 0$ and $\llbracket \mathbf{c}_n \rrbracket(X_1, \dots, X_n) = \sum_{i=1}^n X_i + 1$ for $n \geq 1$, we can encode every natural number n by n compositions of the shape $\underline{n} = \mathbf{c}_1(\dots \mathbf{c}_1(\mathbf{c}_0) \dots)$ and we can encode the identity polynomial by adding the following rule:

$$\text{id}(\mathbf{x}) \rightarrow \text{id}(\text{id}(\mathbf{x}))$$

One can check that the corresponding inequality constraints its quasi-interpretation to be equal to $\llbracket \text{id} \rrbracket(X) = X$ over \mathbb{N} . Moreover we can add arbitrary rules of the shape:

$$\begin{aligned} & \text{id}(\mathbf{c}_n(\mathbf{x}, \dots, \mathbf{x})) \rightarrow \mathbf{f}_n(\mathbf{x}) \\ & \text{id}(\mathbf{c}_0) \rightarrow \mathbf{f}_n(\mathbf{c}_0, \dots, \mathbf{c}_0) \\ & \mathbf{f}_n(\mathbf{c}_1(\mathbf{x})) \rightarrow \underbrace{\mathbf{c}_1(\dots(\mathbf{c}_1(\mathbf{f}_n(\mathbf{x}))\dots))}_{n \text{ times}} \end{aligned}$$

²This will be shown explicitly in the next Subsection.

in order to force the following interpretation $\llbracket \mathbf{f}_n \rrbracket(X) = n \times X$. In the same spirit we can encode addition by:

$$\begin{aligned} \text{id}(c_2(\mathbf{x}, \mathbf{y})) &\rightarrow \text{add}(\mathbf{x}, \mathbf{y}) \\ \text{add}(c_1(\mathbf{x}), c_1(\mathbf{y})) &\rightarrow c_1(c_1(\text{add}(\mathbf{x}, \mathbf{y}))) \end{aligned}$$

in order to force $\llbracket \text{add} \rrbracket(X, Y) = X + Y$ and we can encode multiplication by:

$$\begin{aligned} \mathbf{f}_n(\mathbf{x}) &\rightarrow \text{mult}(\mathbf{x}, \underline{n}) \\ \mathbf{f}_n(\mathbf{x}) &\rightarrow \text{mult}(\underline{n}, \mathbf{x}) \\ \text{mult}(c_1(\mathbf{x}), \mathbf{y}) &\rightarrow \text{add}(\mathbf{y}, \text{mult}(\mathbf{x}, \mathbf{y})) \end{aligned}$$

in order to force $\llbracket \text{mult} \rrbracket(X, Y) = X \times Y$. We let the reader check that this reasoning can be generalized to any degree. Finally, if \mathbf{f} is the symbol whose interpretation has been forced to encode the polynomial P^2 , we add the rule:

$$\mathbf{f}(\mathbf{x}_1, \dots, \mathbf{x}_n) \rightarrow c_1(c_0)$$

to encode the inequality $P^2 \geq 1$. Finally, let us remark that the same (but more technical) kind of encoding can be performed over \mathbb{Q}^+ . \square

Since the encoding presented in the proof of previous Theorem does not depend on the use of a max operator we obtain the following corollary:

Corollary 2. *The QI synthesis problem is undecidable over $\mathbb{N}[\overline{X}]$ and $\mathbb{Q}^+[\overline{X}]$.*

5.4. Decidable synthesis over $\text{Max-Poly}\{\mathbb{R}^+\}$

In order to get a precise upper bound, we define two notions of degree. The first notion, called \times -degree, corresponds to the maximal power of a polynomial whereas the second notion, called max-degree, corresponds to the maximal arity of the max function.

Definition 12 (Degrees). *Given a function³ $Q \neq 0 \in \text{MaxPoly}\{\mathbb{K}\}$ of arity n and normal form $\max(P_1, \dots, P_k)$, with P_i polynomials, then the max-degree of Q is equal to k .*

Moreover, if P_i is a polynomial of degree d_i , where the degree of a n -ary polynomial of the shape $\sum_{l=1}^k \alpha_l X_1^{i_1^l} X_2^{i_2^l} \dots X_n^{i_n^l}$, with $\forall l \in [1, k], \alpha_l \neq 0$, is equal to $\max_{l \in [1, k]} (\sum_{j=1}^n i_j^l)$, then the \times -degree is equal to $\max_{i \in [1, k]} d_i$.

These notions of degree are extended to assignments, the degree of an assignment being the maximal degree of a polynomial in its image.

Definition 13. *The assignment $\llbracket - \rrbracket \in \text{MaxPoly}\{\mathbb{K}\}$ is in $\text{MaxPoly}^{(k, d)}\{\mathbb{K}\}$ if its \times -degree and its max-degree are respectively bounded by the constants d and k .*

³The polynomial 0 will have degrees equal to 0.

Given an assignment $(-)\in \text{MaxPoly}^{(k,d)}\{\mathbb{R}^+\}$ and a function symbol \mathbf{f} of arity n such that \mathbf{f} is in the definition domain of $(-)$. By Proposition 1, the assignment of \mathbf{f} can be written as follows:

$$(\mathbf{f})(\overline{X}) = \max(P[\mathbf{f}, 1](\overline{X}), \dots, P[\mathbf{f}, k](\overline{X}))$$

where $\overline{X} = X_1, \dots, X_n$ and $P[\mathbf{f}, i]$ are polynomials of degree at most d . In other words:

$$P[\mathbf{f}, i](\overline{X}) = \sum a[\mathbf{f}, i, j_1, \dots, j_n] X_1^{j_1} \times \dots \times X_n^{j_n}$$

with $1 \leq i \leq k$ and $\sum_{\ell=1}^n j_\ell \leq d$ and where the variable $a[\mathbf{f}, i, j_1, \dots, j_n] \in \mathbb{R}^+$. Now we show some intermediate lemmata:

Lemma 2 (Subterm encoding). *Given \mathbf{f} of arity n and an assignment $(-)\in \text{MaxPoly}^{(k,d)}\{\mathbb{R}^+\}$ such that (\mathbf{f}) is defined, the subterm property can be encoded by the following first order formula:*

$$S[\mathbf{f}] = \bigwedge_{j \in [1, n]} S[\mathbf{f}, j]$$

with $S[\mathbf{f}, j] = \forall X_1, \dots, X_n, \bigvee_{i \in [1, k]} P[\mathbf{f}, i](\overline{X}) \geq X_j$.

In other words, $S[\mathbf{f}]$ if and only if (\mathbf{f}) is subterm.

Proof. (\mathbf{f}) is subterm iff $\forall X_1, \dots, X_n, (\mathbf{f})(X_1, \dots, X_n) \geq \max(X_1, \dots, X_n)$ iff $\max(P[\mathbf{f}, 1](\overline{X}), \dots, P[\mathbf{f}, k](\overline{X})) \geq \max(X_1, \dots, X_n)$ which is equivalent to $S[\mathbf{f}]$, by Proposition 2. \square

Lemma 3 (Monotonicity encoding). *Given $\mathbf{f} \in \mathcal{D}$ of arity n and an assignment $(-)\in \text{MaxPoly}^{(k,d)}\{\mathbb{R}^+\}$ such that (\mathbf{f}) is defined, the monotonicity property can be encoded by the following first order formula:*

$$M[\mathbf{f}] = \forall X_1, \dots, X_n, \forall Y_1, \dots, Y_n, \\ \bigwedge_{l \in [1, n]} X_l \geq Y_l \implies \bigwedge_{j \in [1, k]} \bigvee_{i \in [1, k]} P[\mathbf{f}, i](\overline{X}) \geq P[\mathbf{f}, j](\overline{Y})$$

In other words, $M[\mathbf{f}]$ if and only if (\mathbf{f}) is monotonic.

Proof. The proof is just an application of Proposition 2. \square

Now we relate the degrees of an expression interpretation with respect to the degree of its symbol interpretations. The main reason for doing so is that we need to encode expression interpretations and not only symbol interpretation in order to encode the rewrite rules of a program.

Proposition 3. *Given $(-)\in \text{MaxPoly}^{(k,d)}\{\mathbb{R}^+\}$ and a term t , we have $(t) \in \text{MaxPlus}^{(k^{|t|}, d^{|t|})}\{\mathbb{R}^+\}$.*

Proof. By induction on the size of a term t . \square

Proposition 3 shows that polynomials can be extended to terms. We write:

$$\langle t \rangle(\bar{X}) = \max(P[t, 1](\bar{X}), \dots, P[t, k'](\bar{X}))$$

for some $k' \leq k^{|t|}$ and with $P[t, j]$ polynomials of degree bounded by $d^{|t|}$, whenever the considered assignment is of max-degree k and \times -degree d .

Lemma 4 (Rule encoding). *Given a TRS $\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle$ and an assignment $\langle - \rangle \in \text{MaxPoly}^{(k, d)}\{\mathbb{R}^+\}$, for each rule $l \rightarrow_{\mathcal{R}} r$, each inequality can be encoded by:*

$$R[l \rightarrow r] = \forall X_1, \dots, X_n, \bigwedge_{j \in [1, l]} \bigvee_{i \in [1, n]} P[l, i](\bar{X}) \geq P[r, j](\bar{X})$$

with $n \leq k^{|l|}$ and $l \leq k^{|r|}$.

In other words, $R[l \rightarrow r]$ if and only if $\langle l \rangle \geq \langle r \rangle$ is satisfied.

Proof. By combining Propositions 2 and 3. □

Proposition 4 (QI encoding). *Given a TRS $\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle$, whose symbols have maximal arity n , define the first order formula:*

$$QI[\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle] = \exists a[\mathbf{f}, i, j_1, \dots, j_n] \in \mathbb{R}^+, \left(\bigwedge_{\mathbf{g} \in \mathcal{D}} (S[\mathbf{g}] \wedge M[\mathbf{g}]) \right) \wedge \left(\bigwedge_{l \rightarrow_{\mathcal{R}} r \in \mathcal{R}} R[l \rightarrow r] \right)$$

$QI[\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle]$ is true if and only if there is an assignment $\langle - \rangle$ that is a quasi-interpretation of $\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle$.

Proof. All the properties of QI are satisfied by Lemmata 2, 3, and 4 □

Theorem 16. $\forall k, d \in \mathbb{N}$, the QI synthesis problem is decidable in exponential time (in the size of the program) over $\text{MaxPoly}^{(k, d)}\{\mathbb{R}^+\}$.

Proof. Given a TRS $\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle$, by Proposition 4, the QI synthesis problem can be turned into checking the satisfaction of the formula $QI[\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle]$. Note that we can extrude all the quantifiers of the formula $QI[\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle]$, after a careful α -conversion, obtaining a new formula under prenex normal form with only one alternation between a block of existential quantifiers (encoding the polynomials multiplicative coefficients) and one block of universal quantifiers (encoding program variables) and we apply the same reasoning than in Theorem 13 (using Theorem 12 again). Note that the exponential upper bound lies in the fact that there are only two blocks of quantifiers ($m = 2$). □

Corollary 3. *The QI synthesis problem is decidable in exponential time over $\mathbb{R}^+[\bar{X}]$.*

5.5. Another interest in the use of reals

The interest of considering quasi-interpretations over the reals does not only rely on the decidability result of Theorem 16. Indeed, we have an analog result to Theorem 8 over MaxPoly quasi-interpretations. It states that there exist programs that do not have any quasi-interpretation over $\text{MaxPoly}\{\mathbb{Q}^+\}$ and, a fortiori $\text{MaxPoly}\{\mathbb{N}\}$, but that admit a quasi-interpretation over $\text{MaxPoly}\{\mathbb{R}^+\}$.

Theorem 17. *There are TRS having a quasi-interpretation over $\text{MaxPoly}\{\mathbb{R}^+\}$, whereas they do not have any quasi-interpretation over $\text{MaxPoly}\{\mathbb{Q}^+\}$.*

Proof. We build such a TRS in order to enforce its quasi-interpretation $\llbracket - \rrbracket \in \text{MaxPoly}\{\mathbb{R}^+\}$ to have an irrational coefficient. Our proof is based on additive QI but we claim that there is a similar proof for the general case. The existence of an infinite number of such TRS follows since we can add infinitely many rules with fresh function symbols on such a program. Moreover we may add the following rule:

$$\text{id}(x) \rightarrow \text{id}(\text{id}(x))$$

It enforces the function symbol id to have a quasi-interpretation of the shape $\llbracket \text{id} \rrbracket(X) = X$ (otherwise if $\llbracket \text{id} \rrbracket(X) > X$, we have $\llbracket \text{id} \rrbracket(\llbracket \text{id} \rrbracket(X)) > \llbracket \text{id} \rrbracket(X)$ and there is no QI for such a program). Consider a fresh 2-ary function symbol \mathbf{g} , a 0-ary constructor symbol $\mathbf{0}$ and a 2-ary constructor symbol \mathbf{c} such that: $\llbracket \mathbf{0} \rrbracket = 0$ and $\llbracket \mathbf{c} \rrbracket(X, Y) = X + Y + 1$. Consider the following rule:

$$\text{id}(\mathbf{0}) \rightarrow \mathbf{g}(\mathbf{0}, \mathbf{0})$$

If $\llbracket - \rrbracket$ is a quasi-interpretation of the TRS then the following inequality holds:

$$0 \geq \llbracket \mathbf{g} \rrbracket(\mathbf{0}, \mathbf{0})$$

Now consider adding the rule:

$$\text{id}(\mathbf{c}(\mathbf{c}(y, y), \mathbf{c}(y, y))) \rightarrow \mathbf{g}(\mathbf{c}(\mathbf{0}, \mathbf{0}), y)$$

$\llbracket - \rrbracket$ has to satisfy that:

$$4 \times Y + 3 \geq \llbracket \mathbf{g} \rrbracket(1, Y)$$

Consequently, $\llbracket \mathbf{g} \rrbracket(X, Y)$ has a \times -degree at most 1 in Y . Otherwise, for an arbitrary large Y , the above inequality is no longer satisfied. Consequently, there is a set I of indexes and polynomials R_i and S_i such that $\llbracket \mathbf{g} \rrbracket(X, Y) = \max_{i \in I} (R_i(X) \times Y + S_i(X))$ and $\llbracket \mathbf{g} \rrbracket(\mathbf{0}, \mathbf{0}) = \max_{i \in I} (S_i(\mathbf{0})) = 0$.

Now consider two 1-ary fresh constructor symbols \mathbf{a} and \mathbf{b} such that $\llbracket \mathbf{a} \rrbracket(X) = X + k$ and $\llbracket \mathbf{b} \rrbracket(X) = X + k'$, for some $k, k' \in \mathbb{N}$. Finally, add the following rules:

$$\text{id}(\mathbf{b}(\mathbf{b}(\mathbf{0}))) \rightarrow \mathbf{g}(\mathbf{0}, \mathbf{g}(\mathbf{0}, \mathbf{b}(\mathbf{0})))$$

$$\text{id}(\mathbf{b}(\mathbf{0})) \rightarrow \mathbf{g}(\mathbf{0}, \mathbf{a}(\mathbf{0}))$$

$$\mathbf{g}(\mathbf{0}, \mathbf{a}(\mathbf{0})) \rightarrow \mathbf{b}(\mathbf{0})$$

$$\mathbf{g}(\mathbf{0}, \mathbf{b}(\mathbf{0})) \rightarrow \mathbf{a}(\mathbf{a}(\mathbf{0}))$$

All these rules correspond to the following inequalities:

$$\begin{aligned} 2 \times k' &\geq \max_{i \in I} (R_i(0))^2 \times k' \\ k' &\geq \max_{i \in I} (R_i(0)) \times k \\ \max_{i \in I} (R_i(0)) \times k &\geq k' \\ \max_{i \in I} (R_i(0)) \times k' &\geq 2 \times k \end{aligned}$$

The first inequality guarantees that $2 \geq \max_{i \in I} (R_i(0))^2$ since $k' \geq 1$. We deduce from second and third inequalities that $k' = \max_{i \in I} (R_i(0)) \times k$. Substituting $\max_{i \in I} (R_i(0)) \times k$ to k' in the last inequality, we obtain $\max_{i \in I} (R_i(0))^2 \times k \geq 2 \times k$ and, consequently, $\max_{i \in I} (R_i(0))^2 \geq 2$, since $k \geq 1$. Finally, $\max_{i \in I} (R_i(0)) = \sqrt{2}$ and the program only admits irrational quasi-interpretations. In particular, it admits the following quasi-interpretation: $\llbracket 0 \rrbracket = 0$, $\llbracket \mathbf{a} \rrbracket (X) = X + 1$, $\llbracket \mathbf{b} \rrbracket (X) = X + \sqrt{2}$, $\llbracket \mathbf{c} \rrbracket (X, Y) = X + Y + 1$, $\llbracket \mathbf{id} \rrbracket (X) = X$ and $\llbracket \mathbf{g} \rrbracket (X, Y) = \max(\sqrt{2}(X + 1)Y, X, Y)$. \square

5.6. The QI synthesis problem over MaxPlus

5.6.1. NP-hardness results

The complexity of the QI synthesis problem over MaxPoly encourage us to consider smaller function sets. In this perspective, Amadio [18] has considered assignments in $\text{MaxPlus}\{\mathbb{N}\}$ ⁴. He has demonstrated that the QI synthesis problem is still a hard problem even on such a small set of functions.

Definition 14. Let $\text{MaxPlus}\{\mathbb{K}\}$ be the set of functions obtained using constants and variables ranging over \mathbb{K} and arbitrary compositions of the operators $+$ and \max .

Now we state a normalization result that is just a corollary of Proposition 1:

Proposition 5. Each function $Q \in \text{MaxPlus}\{\mathbb{K}\}$, $Q \neq 0$, can be written into the following normal form:

$$Q(X_1, \dots, X_n) = \max_{i \in I} \left(\sum_{j=1}^n \alpha_{i,j} X_j + a_i \right)$$

for some finite set of indexes $I \subset \mathbb{N}$ and coefficients $\alpha_{i,j}, a_i \in \mathbb{K}$, $\forall i \in I, j \in [1, n]$.

Theorem 18 (Amadio [18]). *The additive QI synthesis problem is NP-hard over $\text{MaxPlus}\{\mathbb{N}\}$.*

⁴Indeed Amadio considers polynomials with variables and additive coefficients over \mathbb{Q}^+ but with multiplicative coefficients over \mathbb{N} , consequently restricting the shape of allowed interpretations, whereas we will explicitly consider all coefficients in \mathbb{Q}^+ when referring to $\text{MaxPlus}\{\mathbb{Q}^+\}$. Also note that real numbers are not considered in Amadio's result.

In what follows, we will show that the QI synthesis problem remains NP-hard over $\text{MaxPlus}\{\mathbb{R}^+\}$. One could have expected a better result by a naive analogy with linear programming that is P-complete over \mathbb{R}^+ and NP-complete over \mathbb{N} . This result is inspired by the NP-hardness proof suggested in Amadio [18]. However since the quasi-interpretation coefficients are ranging over \mathbb{R}^+ instead of \mathbb{N} it generates some technical encoding problems. Indeed, properties of the shape “If $x + y = 1$ then either $x = 1$ and $y = 0$ or the converse” hold over \mathbb{N} but not over \mathbb{R}^+ . More constraints are thus needed on the considered TRS to encode a reduction from a NP-complete problem.

Theorem 19. *The additive quasi-interpretation synthesis problem is NP-hard over $\text{MaxPlus}\{\mathbb{R}^+\}$.*

The complete proof with key-ingredients is in the Subsection 5.6.2. It proceeds by reducing a 3-CNF problem into a synthesis problem for $\text{MaxPlus}\{\mathbb{R}^+\}$. The reduction follows Amadio [18]. The main difference is that the property $\sum_{j=1}^n \alpha_{i,j} = 1 \Rightarrow (\exists j \text{ such that } \alpha_{i,j} = 1 \text{ and } \forall k \neq j \alpha_{i,k} = 0)$ holds over \mathbb{N} but no longer holds over reals or rationals. We overcome this problem by adding new rules that give sufficient constraints on the considered assignments to allow us to recover such a property. The end of our proof follows Amadio’s proof that encodes literals into a synthesis problem: a function symbol \mathbf{f}_i having a quasi-interpretation $\langle \mathbf{f}_i \rangle = \alpha_1 X_1 + \alpha_2 X_2$ satisfying $(\alpha_1 = 1 \text{ and } \alpha_2 = 2)$ or $(\alpha_1 = 2 \text{ and } \alpha_2 = 1)$ is associated to each literal \mathbf{x}_i of a 3-CNF formula ϕ . We suppose that some fixed constant $k \geq 1$ (respectively $2k$) is the additive constant corresponding to the interpretation of a constructor symbol \mathbf{c} and is an encoding of the truth value **True** (resp. **False**). If the first literal of a disjunction D in ϕ is \mathbf{x}_i , we associate inputs $(\mathbf{c}(0), 0)$ to the function symbol \mathbf{f}_i . In this case, we have $\langle \mathbf{f}_i(\mathbf{c}(0), 0) \rangle = \alpha_1 \times k$ and $\langle \mathbf{f}_i \rangle$ will correspond to **True** if and only if $\alpha_1 = 1$, that is $\langle \mathbf{f}_i \rangle(X_1, X_2) = X_1 + 2 \times X_2$. If the first literal of D is $\neg \mathbf{x}_i$, we associate inputs $(0, \mathbf{c}(0))$ to the function symbol \mathbf{f}_i . In this case, we have $\langle \mathbf{f}_i(\mathbf{c}(0), 0) \rangle = \alpha_2 \times k$ and $\langle \mathbf{f}_i \rangle$ will correspond to **True** if and only if $\alpha_2 = 1$, that is $\langle \mathbf{f}_i \rangle(X_1, X_2) = 2 \times X_1 + X_2$. Finally we require, using constraints (generated by fresh rules) on the QI, that at least one literal (or its negation) is evaluated to k in each disjunction of ϕ by requiring that at most 2 literals of each disjunction are evaluated to **False**. The provided reduction is polynomial in the size of the formula ϕ .

Corollary 4. *The additive quasi-interpretation synthesis problem is NP-hard over $\text{MaxPlus}\{\mathbb{Q}^+\}$.*

Proof. Just notice that the proof presented in Subsection 5.6.2 also holds on \mathbb{Q}^+ . \square

5.6.2. Proof of NP-hardness over $\text{MaxPlus}\{\mathbb{R}^+\}$

In this section, we show the NP-hardness of the synthesis problem over $\text{MaxPlus}\{\mathbb{R}^+\}$ by exhibiting a reduction of every 3-CNF formula satisfiability problem into a quasi-interpretation synthesis problem. For that purpose, we need some intermediate and technical propositions.

Proposition 6. *Given a TRS $\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle$ having a quasi-interpretation $\llbracket - \rrbracket \in \text{MaxPlus}\{\mathbb{R}^+\}$. For every $\mathbf{f} \in \mathcal{D}$ such that $\llbracket \mathbf{f} \rrbracket(X_1, \dots, X_n) = \max_{i \in I} (\sum_{j=1}^n \alpha_{i,j} \times X_j + a_i)$ we have:*

$$\forall j \leq n, \exists i \in I, \alpha_{i,j} \geq 1$$

Proof. Suppose that $\exists j \leq n, \forall i \in I, \alpha_{i,j} < 1$ holds and let j_0 be the value of index j on which it holds.

Now take the particular values $x_k = 0, \forall k \neq j_0$ and $x_{j_0} > \max_{i \in I} (a_i / (1 - \alpha_{i,j_0}))$ we have:

$$\begin{aligned} \llbracket \mathbf{f} \rrbracket(x_1, \dots, x_{j_0}, \dots, x_n) &= \max_{i \in I} (\alpha_{i,j_0} \times x_{j_0} + a_i) \\ &< \max_{i \in I} (\alpha_{i,j_0} \times x_{j_0} + (1 - \alpha_{i,j_0}) \times x_{j_0}) \\ &< x_{j_0} \end{aligned}$$

Note that x_{j_0} is clearly defined since $\forall i \in I, \alpha_{i,j_0} < 1$. Consequently, this contradicts the subterm property stating that $\forall j \leq n, \forall X_j \in \mathbb{R}^+, \llbracket \mathbf{f} \rrbracket(X_1, \dots, X_n) \geq X_j$. \square

Proposition 7. *There exist a TRS $\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle$ and a function symbol $\mathbf{f} \in \mathcal{D}$ such that if $\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle$ has an additive quasi-interpretation $\llbracket - \rrbracket \in \text{MaxPlus}\{\mathbb{R}^+\}$ then at least one of the following conditions holds:*

1. $\llbracket \mathbf{f} \rrbracket(X_1, \dots, X_n) = \max(X_1, \dots, X_n)$
2. $\llbracket \mathbf{f} \rrbracket(X_1, \dots, X_n) = \max_{i \in I} (\sum_{j=1}^n \alpha_{i,j} \times X_j)$ (i.e. $\llbracket \mathbf{f} \rrbracket(0, \dots, 0) = 0$)
3. $\llbracket \mathbf{f} \rrbracket(X_1, \dots, X_n) = \sum_{j=1}^n \alpha_{i,j} \times X_j$

Proof. 1. We show the first equality by generating the rules of \mathcal{R} in order to constraint the quasi-interpretation of \mathbf{f} . Suppose that \mathbf{f} admits a quasi-interpretation of the shape $\llbracket \mathbf{f} \rrbracket(X_1, \dots, X_n) = \max_{i \in I'} (\sum_{j=1}^n \alpha_{i,j} \times X_j + a_i)$ and consider adding the following rule:

$$\mathbf{f}(\mathbf{x}_1, \dots, \mathbf{x}_n) \rightarrow \mathbf{f}(\mathbf{f}(\mathbf{x}_1, \dots, \mathbf{x}_n), \dots, \mathbf{f}(\mathbf{x}_1, \dots, \mathbf{x}_n))$$

If $\llbracket - \rrbracket$ is a quasi-interpretation then it has to satisfy:

$$\llbracket \mathbf{f}(\mathbf{x}_1, \dots, \mathbf{x}_n) \rrbracket \geq \max_{i \in I'} ((\sum_{j=1}^n \alpha_{i,j}) \times \llbracket \mathbf{f}(\mathbf{x}_1, \dots, \mathbf{x}_n) \rrbracket + a_i)$$

Consequently, $\forall i \in I', \sum_{j=1}^n \alpha_{i,j} \leq 1$. Using Proposition 6, we have that for each j there is a particular $i_j \in I'$ such that $\alpha_{i_j,j} \geq 1$. Combined with previous inequality, it implies that $\alpha_{i_j,j} = 1$ and $\forall l, l \neq j, \alpha_{i_j,l} = 0$.

So we can write the quasi-interpretation of \mathbf{f} as follows:

$$\llbracket \mathbf{f} \rrbracket(X_1, \dots, X_n) = \max(X_1 + a_{i_1}, \dots, X_n + a_{i_n}, \max_{i \in I} (\sum_{j=1}^n \alpha_{i,j} \times X_j + a_i))$$

with $I = I' - \{i_1, \dots, i_n\}$ and $\forall j \leq n, \forall i \in I, \alpha_{i,j} < 1$.

For an arbitrarily large value $x \in \mathbb{R}^+$ (take $x > \max_{i \in I} ((a_i - a_{i_1}) / (1 -$

$\alpha_{i,1}$)), we have $\langle \mathbf{f} \rangle(x, 0, \dots, 0) = x + a_{i_1}$, with $a_{i_1} \geq 0$. Indeed $\forall i \in I$, $\alpha_{i,1} \times x + a_i < x + a_{i_1}$. It implies that:

$$\begin{aligned} \langle \mathbf{f} \rangle(x, 0, \dots, 0) &= x + a_{i_1} \\ &\geq \langle \mathbf{f} \rangle(\langle \mathbf{f} \rangle(x, 0, \dots, 0), \dots, \langle \mathbf{f} \rangle(x, 0, \dots, 0)) \\ &\geq \langle \mathbf{f} \rangle(x + a_{i_1}, \dots, x + a_{i_1}) \\ &\geq x + 2 \times a_{i_1} \end{aligned}$$

Consequently, $a_{i_1} = 0$. Since we can perform the same reasoning for each constant a_{i_k} , the quasi-interpretation can be written:

$$\langle \mathbf{f} \rangle(X_1, \dots, X_n) = \max(X_1, \dots, X_n, \max_{i \in I} (\sum_{j=1}^n \alpha_{i,j} \times X_j + a_i))$$

with $\sum_{j=1}^n \alpha_{i,j} \leq 1$. Now consider adding the following rule to the program:

$$\mathbf{f}(\mathbf{b}(\mathbf{x}_1, 0), \dots, \mathbf{b}(\mathbf{x}_n, 0)) \rightarrow \mathbf{f}(\mathbf{b}(\mathbf{x}_1, \mathbf{f}(0, \dots, 0)), \dots, \mathbf{b}(\mathbf{x}_n, \mathbf{f}(0, \dots, 0)))$$

with \mathbf{b} a constructor symbol such that $\langle \mathbf{b} \rangle(X, Y) = X + Y + k_{\mathbf{b}}$, $k_{\mathbf{b}} \geq 1$. In order for $\langle - \rangle$ to be a QI, it is necessary to check that

$$\langle \mathbf{f}(\mathbf{b}(\mathbf{x}_1, 0), \dots, \mathbf{b}(\mathbf{x}_n, 0)) \rangle \geq \langle \mathbf{f}(\mathbf{b}(\mathbf{x}_1, \mathbf{f}(0, \dots, 0)), \dots, \mathbf{b}(\mathbf{x}_n, \mathbf{f}(0, \dots, 0))) \rangle$$

It implies by choosing the particular values $\langle \mathbf{x}_1 \rangle = \dots = \langle \mathbf{x}_n \rangle = x \in \mathbb{R}^+$:

$$\max_{i \in I} (\sum_{j=1}^n \alpha_{i,j} \times (x + k_{\mathbf{b}}) + a_i) \geq \max_{i \in I} (\sum_{j=1}^n \alpha_{i,j} \times (x + k_{\mathbf{b}} + \max_{k \in I} (a_k)) + a_i)$$

Suppose that l is the index for which $\max_{i \in I} (\sum_{j=1}^n \alpha_{i,j})$ is reached. For an arbitrary large x and since $\sum_{j=1}^n \alpha_{l,j} = 1$ and $a_l = 0$, we have $x + k_{\mathbf{b}} \geq x + k_{\mathbf{b}} + \max_{k \in I} (a_k)$. It implies $a_k = 0$, $\forall k \in I$. Finally we have:

$$\langle \mathbf{f} \rangle(X_1, \dots, X_n) = \max(\max(X_1, \dots, X_n), \max_{i \in I} (\sum_{j=1}^n \alpha_{i,j} \times X_j))$$

with $\sum_{j=1}^n \alpha_{i,j} \leq 1$. Since $\forall X_1, \dots, \forall X_n \in \mathbb{R}^+$, $\max(X_1, \dots, X_n) \geq \max_{i \in I} (\sum_{j=1}^n \alpha_{i,j} \times X_j)$ holds, we obtain:

$$\langle \mathbf{f} \rangle(X_1, \dots, X_n) = \max(X_1, \dots, X_n)$$

2. Now we show the second equality. Given \mathbf{g} a function symbol such that $\langle \mathbf{g} \rangle(X_1, \dots, X_n) = \max_{i \in I} (\sum_{j=1}^n \alpha_{i,j} \times X_j + a_i)$. We add the rule:

$$\mathbf{id}(\mathbf{d}(\mathbf{x}_1, \dots, \mathbf{x}_n)) \rightarrow \mathbf{d}(\mathbf{g}(0, \dots, 0), 0, \dots, 0)$$

with \mathbf{id} a function symbol such that $\langle \mathbf{id} \rangle(X) = X$ (There exists such a function symbol by Proposition 7, item (1)) and \mathbf{d} a n -ary constructor symbol such that $\langle \mathbf{d} \rangle(X_1, \dots, X_n) = \sum_{i=1}^n X_i + k_{\mathbf{d}}$, $k_{\mathbf{d}} \geq 1$. The corresponding assignment has to satisfy $k_{\mathbf{d}} + \sum_{j=1}^n X_j \geq k_{\mathbf{d}} + \max_{i \in I} (a_i)$. It implies that $\forall i \in I$, $a_i = 0$. Consequently, $\langle \mathbf{g} \rangle(X_1, \dots, X_n) = \max_{i \in I} (\sum_{j=1}^n \alpha_{i,j} X_j)$.

3. Consider a function symbol \mathbf{f} of arity n . Its quasi-interpretation can be constrained to be of the shape:

$$\langle \mathbf{f} \rangle (X_1, \dots, X_n) = \alpha_1 \times X_1 + \dots + \alpha_n \times X_n$$

by adding the following rules to the program:

$$\text{id}(\mathbf{c}(\mathbf{x})) \rightarrow \mathbf{c}(\mathbf{f}(0, \dots, 0, \mathbf{x}, 0, \dots, 0))$$

with \mathbf{x} appearing at the i -th position in the right hand side of the rule, $\forall i \in [1, n]$, \mathbf{c} a 1-ary constructor symbol such that $\langle \mathbf{c} \rangle (X) = X + k$, $k \geq 1$, and with id a function symbol such that $\langle \text{id} \rangle (X) = X$.

□

Proposition 8. *There exist a TRS $\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle$ and a function symbol $\mathbf{f} \in \mathcal{D}$ of arity 2 such that if $\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle$ has an additive quasi-interpretation $\langle _ \rangle \in \text{MaxPlus}\{\mathbb{R}^+\}$ then the following conditions both hold:*

- $\langle \mathbf{f} \rangle (X_1, X_2) = \alpha_1 \times X_1 + \alpha_2 \times X_2$
- $(\alpha_1 = 1 \wedge \alpha_2 = 2) \vee (\alpha_1 = 2 \wedge \alpha_2 = 1)$

Proof. By Proposition 7, we can enforce a 2-ary function symbol \mathbf{f} to have the following quasi-interpretation by adding arbitrary rules to constraint its quasi-interpretation:

$$\langle \mathbf{f} \rangle (X_1, X_2) = \max_{i \in I} (\alpha_{i,1} \times X_1 + \alpha_{i,2} \times X_2)$$

We define $\alpha_j = \max_{i \in I} (\alpha_{i,j})$, for $j \in \{1, 2\}$, and $\alpha = \max_{i \in I} (\alpha_{i,1} + \alpha_{i,2})$. These constants satisfy the following inequality $\alpha_1 + \alpha_2 \geq \alpha$. Now add the following rule to the considered TRS:

$$\mathbf{f}(\mathbf{b}(\mathbf{x}_1, 0), \mathbf{b}(\mathbf{x}_2, 0)) \rightarrow \mathbf{b}(\mathbf{f}(\mathbf{x}_1, 0), \mathbf{f}(0, \mathbf{x}_2))$$

with \mathbf{b} a 2-ary constructor symbol such that $\langle \mathbf{b} \rangle (X) = X + k$ and $\langle 0 \rangle = 0$. For the particular values $\langle \mathbf{x}_1 \rangle = \langle \mathbf{x}_2 \rangle = x \in \mathbb{R}^+$, the corresponding quasi-interpretation has to satisfy $\alpha \times (x + k) \geq k + (\alpha_1 + \alpha_2) \times x$. Consequently, for an arbitrarily large x , $\alpha = \alpha_1 + \alpha_2$ and we can write:

$$\langle \mathbf{f} \rangle (X_1, X_2) = \alpha_1 \times X_1 + \alpha_2 \times X_2$$

since $\exists j \in I$, $\alpha_{j,1} = \alpha_1$ and $\alpha_{j,2} = \alpha_2$. Indeed it implies that $\forall X_1, X_2 \in \mathbb{R}^+$, $\forall i \in I$ $i \neq j$, $\alpha_{i,1} \times X_1 + \alpha_{i,2} \times X_2 \geq \alpha_{i,1} \times X_1 + \alpha_{i,2} \times X_2$.

By virtue of the subterm condition, $\alpha_1, \alpha_2 \geq 1$. We add new rules over \mathbf{f} in order to constraint α_1 and α_2 to satisfy the following condition $(\alpha_1 = 1 \wedge \alpha_2 = 2) \vee (\alpha_1 = 2 \wedge \alpha_2 = 1)$:

$$\begin{aligned} \mathbf{f}(\mathbf{c}(\mathbf{x}_1), \mathbf{c}(\mathbf{x}_2)) &\rightarrow \mathbf{c}(\mathbf{c}(\mathbf{c}(0))) \\ \text{id}(\mathbf{c}(\mathbf{c}(\mathbf{c}(\mathbf{x})))) &\rightarrow \mathbf{f}(\mathbf{c}(0), \mathbf{c}(0)) \end{aligned}$$

If $\langle - \rangle$ is a quasi-interpretation, \mathbf{c} is a 1-ary constructor symbol such that $\langle \mathbf{c} \rangle(X) = X + k$ and \mathbf{id} is a function symbol such that $\langle \mathbf{id} \rangle(X) = X$ (such a symbol exists by Proposition 7), we deduce from these rules that $\alpha_1 + \alpha_2 = 3$. By adding the rule:

$$\mathbf{id}(\mathbf{c}(\mathbf{c}(x))) \rightarrow \mathbf{f}(\mathbf{f}(0, \mathbf{c}(0)), 0)$$

we check that $2 \times k + x \geq \alpha_1 \times \alpha_2 \times k$. In other words, $2 \geq \alpha_1 \times \alpha_2$. Since $\alpha_1 = 3 - \alpha_2$, we have to check the inequality $\alpha_1^2 - 3 \times \alpha_1 + 2 \geq 0$ with $2 \geq \alpha_1 \geq 1$. The only corresponding solutions are $(\alpha_1 = 1 \wedge \alpha_2 = 2) \vee (\alpha_1 = 2 \wedge \alpha_2 = 1)$. \square

Theorem 19. *The additive quasi-interpretation synthesis problem is NP-hard over $\text{MaxPlus}\{\mathbb{R}^+\}$.*

Proof. We encode the satisfiability of a 3-SAT problem under 3-CNF into a QI synthesis problem. Given a 3-CNF formula ϕ , we generate a TRS $\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle$ such that $\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle$ admits a quasi-interpretation if and only if ϕ is satisfiable. In this perspective, we associate to each literal \mathbf{x}_i appearing in a given 3-CNF formula ϕ , a fresh 2-ary function symbol \mathbf{f}_i and its corresponding rules such that $\langle \mathbf{f}_i \rangle(X_1, X_2) = \alpha_1^i \times X_1 + \alpha_2^i \times X_2$, with $(\alpha_1^i = 1 \wedge \alpha_2^i = 2) \vee (\alpha_1^i = 2 \wedge \alpha_2^i = 1)$. Note that this is made possible by Proposition 8.

The table of Figure 2 subsumes the distinct values taken by the quasi-interpretation $\langle \mathbf{f}_i \rangle$ wrt its coefficients and its inputs, for some constructor symbols \mathbf{c} and 0 such that $\langle \mathbf{c} \rangle(X) = X + k$ and $\langle 0 \rangle = 0$.

Coefficients of $\langle \mathbf{f}_i \rangle$: (α_1^i, α_2^i)	inputs: $(\mathbf{x}_1, \mathbf{x}_2)$	Value of: $\langle \mathbf{f}_i(\mathbf{x}_1, \mathbf{x}_2) \rangle$
(1,2)	$(\mathbf{c}(0), 0)$	k
(1,2)	$(0, \mathbf{c}(0))$	$2 \times k$
(2,1)	$(\mathbf{c}(0), 0)$	$2 \times k$
(2,1)	$(0, \mathbf{c}(0))$	k

Figure 2: Values of $\langle \mathbf{f}_i \rangle$ wrt its coefficients and inputs

Let the constant k (respectively $2 \times k$) encode the truth value **True** (respectively **False**). If a literal corresponds to **True** (resp. **False**) then we will encode this information by constraining \mathbf{f}_i to have a quasi-interpretation equal to $X_1 + 2 \times X_2$ (resp. $2 \times X_1 + X_2$).

Given a disjunction D of the formula ϕ , there are two possibilities:

- (i) If the first literal of D is \mathbf{x}_i , we associate inputs $(\mathbf{c}(0), 0)$ to the function symbol \mathbf{f}_i . In this case, we have $\langle \mathbf{f}_i(\mathbf{c}(0), 0) \rangle = \alpha_1^i \times k$ and $\langle \mathbf{f}_i \rangle$ will correspond to **True** if and only if $\alpha_1^i = 1$, that is $\langle \mathbf{f}_i \rangle(X_1, X_2) = X_1 + 2 \times X_2$.

- (ii) If the first literal of D is $\neg x_i$, we associate inputs $(0, c(0))$ to the function symbol \mathbf{f}_i . In this case, we have $\langle \mathbf{f}_i(c(0), 0) \rangle = \alpha_2^i \times k$ and $\langle \mathbf{f}_i \rangle$ will correspond to **True** if and only if $\alpha_2^i = 1$, that is $\langle \mathbf{f}_i \rangle(X_1, X_2) = 2 \times X_1 + X_2$.

Using the notation $\phi_D^{\mathbf{x}}$ to represent the arguments of the function symbol encoding \mathbf{x} in the disjunction D , we have:

$$\phi_D^{\mathbf{x}} = \begin{cases} c(0), 0 & \text{if } \mathbf{x} \text{ appears in } D \\ 0, c(0) & \text{if } \neg \mathbf{x} \text{ appears in } D \end{cases}$$

$\langle \mathbf{f}(\phi_D^{\mathbf{x}}) \rangle$ is equal to k if $\langle \mathbf{f} \rangle$ corresponds to **True** and \mathbf{x} appears in D or $\langle \mathbf{f} \rangle$ corresponds to **False** and $\neg \mathbf{x}$ appears in D .

$\langle \mathbf{f}(\phi_D^{\mathbf{x}}) \rangle$ is equal to $2 \times k$ if $\langle \mathbf{f} \rangle$ corresponds to **True** and $\neg \mathbf{x}$ appears in D or $\langle \mathbf{f} \rangle$ corresponds to **False** and \mathbf{x} appears in D .

It remains to encode disjunctions: To each disjunction D in the formula ϕ and containing literals \mathbf{x}_i , \mathbf{x}_j and \mathbf{x}_l , we associate the following rule:

$$\mathbf{id}(c(c(c(c(c(\mathbf{x})))))) \rightarrow \mathbf{f}(\mathbf{f}_i(\phi_D^{\mathbf{x}_i}), \mathbf{f}_j(\phi_D^{\mathbf{x}_j}), \mathbf{f}_l(\phi_D^{\mathbf{x}_l}))$$

\mathbf{f} and \mathbf{id} being symbols defined by rewrite rules such that their quasi-interpretations are defined by $\langle \mathbf{id} \rangle(X) = X$ and $\langle \mathbf{f} \rangle(X_1, X_2, X_3) = \alpha_1 \times X_1 + \alpha_2 \times X_2 + \alpha_3 \times X_3$. Note that such symbols exist by items (1) and (3) of Proposition 7. Moreover, by Proposition 6, $\alpha_1, \alpha_2, \alpha_3 \geq 1$. The quasi-interpretation of the obtained TRS has to satisfy:

$$5 \times k + X \geq \langle \mathbf{f}_i(\phi_D(\mathbf{x}_i)) \rangle + \langle \mathbf{f}_j(\phi_D(\mathbf{x}_j)) \rangle + \langle \mathbf{f}_l(\phi_D(\mathbf{x}_l)) \rangle$$

This inequality enforces at least one of the $\langle \mathbf{f}_p(\phi_D(\mathbf{x}_p)) \rangle$ (for $p \in \{i, j, l\}$) to have value k (i.e. to be **True**) and enforces at most two to have value $2k$. Otherwise it is not satisfied because $\neg(5 \times k \geq 6 \times k)$. We encode in the same spirit all the disjunctions of ϕ . Every assignment satisfying ϕ will clearly correspond to the existence of a suitable quasi-interpretation for the program. Indeed, just take $\langle \mathbf{f}_i \rangle(X_1, X_2) = X_1 + 2 \times X_2$ (resp. $2 \times X_1 + X_2$) for each literal assigned to **True** (resp. **False**). Conversely, if the program admits a quasi-interpretation then every disjunction maybe evaluated to true by assigning the truth value **True** to each literal corresponding to a quasi-interpretation of the shape $X_1 + 2 \times X_2$. Finally, we have encoded a 3-CNF problem into a QI synthesis problem over $\text{MaxPlus}\{\mathbb{R}^+\}$ using a polynomial time reduction. Indeed the number of added rules is linear in the the size the formula since each intermediate proposition only introduce a constant number of new rules in the considered TRS. Consequently, this problem is NP-hard. \square

5.6.3. NP-completeness over MaxPlus

After studying NP-hardness results over MaxPlus , we are interested in completeness results on this function space. We start to introduce the first result demonstrated by Amadio in [18] over $\text{MaxPlus}\{0, 1\}$. Let $\text{MaxPlus}\{0, 1\}$ be

the set of functions obtained using constants ranging over $\{0, 1\}$ and variables ranging over \mathbb{Q}^+ and arbitrary compositions of the operators $+$ and \max ⁵.

Theorem 20 (Amadio [18]). *The additive QI synthesis problem is NP-complete over $\text{MaxPlus}\{0, 1\}$.*

We try to extend this result to \mathbb{N} and \mathbb{Q}^+ . For that purpose, we focus on the *QI verification problem* that consists in checking that an assignment of a given TRS is a quasi-interpretation. We show that this problem can be solved in polynomial time over MaxPlus if we consider assignment of max-degree k and $+$ -degree d polynomially bounded by the TRS size. This is not a restrictive condition since most of the TRS admitting a quasi-interpretation in MaxPlus satisfy it. Indeed arity of the max is indexed by the number of rules in the TRS. Each rule may create a new constraint and may consequently increase the max arity by 1. Finally, the arity of the $+$ -degree is trivially indexed by the size of expressions in the rules.

Definition 15 ($+$ -degree and max-degree). *Given a function Q of arity n in $\text{MaxPlus}\{\mathbb{K}\}$ of normal form $\max(P_1, \dots, P_m)$ with:*

$$P_i = \sum_{j \in [1, n]} \alpha_{i, j} \times X_j + \alpha_{i, 0}$$

its $+$ -degree is equal to $\max_{j \in [0, n], i \in [1, m]} \alpha_{i, j}$. In other words, the $+$ -degree of Q is its greatest multiplicative coefficient. Its max-degree is equal to m .

Definition 16. *Let $\text{MaxPlus}^{(k, d)}\{\mathbb{K}\}$ be the set of $\text{MaxPlus}\{\mathbb{K}\}$ functions of $+$ -degree bounded by the constant d and max-degree bounded by the constant k . Given a TRS $\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle$ and an assignment $\langle \cdot \rangle, \langle \cdot \rangle \in \text{MaxPlus}^{(k, d)}\{\mathbb{K}\}$ if:*

$$\forall l \rightarrow r \in \mathcal{R}, \langle l \rangle, \langle r \rangle \in \text{MaxPlus}^{(k, d)}\{\mathbb{K}\}$$

Theorem 21 (Verification). *Given a TRS $\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle$ and an assignment $\langle \cdot \rangle \in \text{MaxPlus}^{(k, d)}\{\mathbb{R}^+\}$, we can check in polynomial time in d and k that $\langle \cdot \rangle$ is a quasi-interpretation of $\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle$.*

Proof. Given a TRS $\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle$ and an assignment $\langle \cdot \rangle$, for each rule of the shape $l \rightarrow r$, we can compute $\langle l \rangle$ and $\langle r \rangle$ in polynomial time relatively to k and d , by definition of $\text{MaxPlus}^{(k, d)}$ assignments. Consequently, it remains to check that the inequalities of the shape $\langle l \rangle \geq \langle r \rangle$ are satisfied (we also have to check some inequalities for monotonicity and subterm properties that we omit). The total number of such inequalities is polynomially bounded by the TRS size r . Moreover, by Proposition 2, we can eliminate the max operators so that each inequality is transformed into the conjunction and disjunction of k^2 inequalities of the shape $P \geq Q$, with $P, Q \in \text{MaxPlus}\{\mathbb{R}^+\}$. Such inequalities have size polynomially bounded by k and d . We can check their satisfaction in

⁵Such functions were called multi-linear polynomials in [18].

polynomial time in these two parameters using linear programming over \mathbb{R}^+ , iterating this procedure at most $r \times k^2$. \square

Theorem 22. *The additive quasi-interpretation synthesis problem is NP-complete over $\text{MaxPlus}^{(k,d)}\{\mathbb{N}\}$, for $d \geq 2$.*

Proof. The NP-hardness has been demonstrated in Theorem 18. For that purpose, we need a +-degree of at least 2 in our encoding of 3-CNF. We have shown in Theorem 21, that the verification problem that consists in checking for a candidate assignment that it is a quasi-interpretation can be solved in polynomial time (if variables are extended to \mathbb{R}^+). It remains to see that the size of each solution is bounded polynomially by the input size (the TRS size): it is the case since its degrees are bounded by constants k and d . \square

Theorem 23. *Let $\mathbb{Q}_{\leq d}^+$ be the subset of \mathbb{Q}^+ such that every rational has both numerator and denominator bounded by d . The additive quasi-interpretation synthesis problem is NP-complete over $\text{MaxPlus}^{(k,d)}\{\mathbb{Q}_{\leq d}^+\}$, for $d \geq 2$.*

Proof. Every rational from $\mathbb{Q}_{\leq d}^+$ can be encoded by two integers smaller than d and, consequently, has a size bounded polynomially by d . \square

Such a result does not hold in general for \mathbb{R}^+ because of the representation problem in such a space: we do not know how to encode the data since a real number is generally not bounded even if we have bounded degrees.⁶

6. Dependency Pair interpretations

6.1. DP-interpretations as sup-interpretations

The notion of sup-interpretation was introduced in [1] in order to increase the intentionality of interpretation methods. One of the main distinction with quasi-interpretations lies in the subterm property (cf. Definition 9): sup-interpretations do not need to satisfy such a property. Consequently, the subterm property drastically restricts the sup-interpretation space. For example, a function defined by $\mathbf{f}(x, y) \rightarrow x$ has a QI at least equal to $(\mathbf{f})(X, Y) = \max(X, Y)$ whereas one would expect its sup-interpretation to be equal to $\theta(\mathbf{f})(X, Y) = X$ since the second parameter is dropped. To overcome this problem, we introduce a new notion of sup-interpretations, namely DP-interpretations, based on the notion of dependency pair (DP) introduced by Arts and Giesl [6] for showing program termination. Note that a similar notion was introduced in [17] for characterizing FPTIME but was not related to the notion of sup-interpretation. A last point to mention is that DP-interpretations are not a DP-method since they do not ensure termination but rather a method for space analysis inspired by DP-methods. We start by briefly reviewing the notion of dependency pair:

⁶This is Corrigendum to [7] where it was wrongly stated that the QI synthesis problem is NP-complete over $\text{MaxPlus}^{(k,d)}\{\mathbb{R}^+\}$.

Definition 17 (DP). Given a TRS $\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle$, the set of dependency pair symbols \mathcal{D}^\sharp is defined by $\mathcal{D}^\sharp = \mathcal{D} \cup \{\mathbf{f}^\sharp \mid \mathbf{f} \in \mathcal{D}\}$, \mathbf{f}^\sharp being a fresh function symbol of the same arity as \mathbf{f} . Given a term $t = \mathbf{f}(t_1, \dots, t_n)$, let t^\sharp be a notation for $\mathbf{f}^\sharp(t_1, \dots, t_n)$.

A dependency pair is a pair $l^\sharp \rightarrow u^\sharp$ if $u^\sharp = \mathbf{g}^\sharp(t_1, \dots, t_n)$, for some $\mathbf{g} \in \mathcal{D}$, and if there is a context $C[\diamond]$ such that $l \rightarrow C[u] \in \mathcal{R}$ and u is not a proper subterm of l . Let $DP(\mathcal{R})$ be the set of all dependency pairs in $\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle$.

Definition 18 (DP-interpretation). Given a TRS $\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle$, a (additive) DP-interpretation (DPI for short) is a monotonic (additive) assignment $\llbracket - \rrbracket$ over \mathbb{K} extended to \mathcal{D}^\sharp by $\forall \mathbf{f}, \in \mathcal{D}, \llbracket \mathbf{f}^\sharp \rrbracket = \llbracket \mathbf{f} \rrbracket$ and which satisfies:

1. $\forall l \rightarrow r \in \mathcal{R}, \llbracket l \rrbracket \geq \llbracket r \rrbracket$
2. $\forall l^\sharp \rightarrow u^\sharp \in DP(\mathcal{R}), \llbracket l^\sharp \rrbracket \geq \llbracket u^\sharp \rrbracket$

where the DP-interpretation $\llbracket - \rrbracket$ is extended canonically to terms as usual.

Notice that the main distinction with QI is that the subterm property has been replaced by Condition 2. We obtain a result similar to Theorem 14:

Theorem 24. Given a TRS $\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle$ having an additive DP-interpretation $\llbracket - \rrbracket$ then $\llbracket - \rrbracket$ is a sup-interpretation.

Moreover, we can show that every quasi-interpretation is a DP-interpretation.

Theorem 25. Given a TRS $\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle$ having a quasi-interpretation $\langle \llbracket - \rrbracket, \langle - \rangle \rangle$ is a DP-interpretation.

Proof. By Definition 9, $\langle - \rangle$ is a monotonic assignment which satisfies $\forall l \rightarrow r \in \mathcal{R}, \langle l \rangle \geq \langle r \rangle$. Now, take $s^\sharp \rightarrow t^\sharp \in DP(\mathcal{R})$. By definition, there is a context $C[\diamond]$ such that $s \rightarrow_{\mathcal{R}} C[t] \in \mathcal{R}$. For each term t , $\langle C[t] \rangle \geq \langle t \rangle$ since $\langle C[t] \rangle$ is obtained by composition of subterm functions (the subterm property is stable by composition). Consequently, $\langle s^\sharp \rangle = \langle s \rangle \geq \langle C[t] \rangle \geq \langle t \rangle = \langle t^\sharp \rangle$. \square

As expected, the converse property does not hold. There are TRS that admit an (additive) DP-interpretation but no (additive) quasi-interpretation, as illustrated by the following example:

Example 4.

$$\begin{array}{ll} \text{half}(0) \rightarrow 0 & \text{half}(1) \rightarrow 0 \\ \text{half}(x+2) \rightarrow \text{half}(x) + 1 & \\ \log(x+2) \rightarrow \log(\text{half}(x+2)) + 1 & \log(1) \rightarrow 0 \end{array}$$

The above TRS has no additive quasi-interpretation since an additive quasi-interpretation such that $\langle +1 \rangle(X) = X + k$, for $k \geq 1$, would have to satisfy the following inequalities:

$$\begin{aligned} \langle \log(x+2) \rangle &\geq \langle \log(\text{half}(x+2)) + 1 \rangle \\ &\geq \langle \log(\text{half}(x+2)) \rangle + k \geq \langle \log(x+2) \rangle + k \end{aligned}$$

By subterm and monotonicity properties. However, we let the reader check that it admits the following additive DP-interpretation $\llbracket 0 \rrbracket = 1$, $\llbracket +1 \rrbracket(X) = X + 1$, $\llbracket \text{half} \rrbracket(X) = (X + 1)/2$ and $\llbracket \text{log} \rrbracket(X) = 2 \times X$.

6.2. Decidability results over $\mathbb{K}[\overline{X}]$, $\text{MaxPoly}\{\mathbb{K}\}$ and $\text{MaxPlus}\{\mathbb{K}\}$

In this section, we review the results of the DPI synthesis problem.

Definition 19 (DPI synthesis problem). *Given a TRS $\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle$, is there an assignment $\llbracket - \rrbracket$ such that $\llbracket - \rrbracket$ is a DP-interpretation of $\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle$?*

Theorem 26. *The DPI synthesis problem is:*

1. *undecidable over $\text{MaxPoly}\{\mathbb{N}\}$ and $\text{MaxPoly}\{\mathbb{Q}^+\}$*
2. *decidable in exponential time over $\text{MaxPoly}^{(k,d)}\{\mathbb{R}^+\}$*
3. *undecidable over $\mathbb{N}[\overline{X}]$ and $\mathbb{Q}^+[\overline{X}]$*
4. *decidable in exponential time over $\mathbb{R}^+[\overline{X}]$*

Proof. (1) is a corollary of Theorem 15. The subterm property is withdrawn and replaced by inequalities on dependency pairs. These inequalities do not change the undecidability of the problem. (2) is also a corollary of Theorem 16 using the same reasoning: the encoding of the subterm property is no longer needed and replaced by the encoding of inequalities on DP. Since the number of DP is at most linear in the size of the program, these new inequalities does not impact the complexity of the algorithm. (3) is a consequence of (1) and (4) is a consequence of (2) because polynomials are functions in MaxPoly . \square

6.3. NP-hardness over $\text{MaxPlus}\{\mathbb{K}\}$

Now we show NP-hardness and NP-completeness results:

Theorem 27. *The additive DPI synthesis problem is:*

1. *NP-hard over $\text{MaxPlus}\{\mathbb{K}\}$, $\mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+, \mathbb{R}^+\}$*
2. *NP-complete over $\text{MaxPlus}^{(k,d)}\{\mathbb{N}\}$, $d \geq 2$*
3. *NP-complete over $\text{MaxPlus}^{(k,d)}\{\mathbb{Q}_{\leq d}^+\}$, $d \geq 2^7$*

Proof. (1) We use the encoding in the proof of Theorem 19. There is just one difficulty to face: By Theorem 25 every QI of a given program is a DPI but the converse does not hold. Consequently, it might be easier to find the DPI of a given program than to find its QI, the solution space being greater. Consequently, we have to enforce that each DPI of the reduction is also a QI. This can be done by adding the following rules to the program, $\forall \mathbf{f} \in \mathcal{D}$ of arity n and $\forall i \in \{1, \dots, n\}$:

$$\mathbf{f}(\mathbf{x}_1, \dots, \mathbf{x}_n) \rightarrow \mathbf{x}_i$$

⁷Cf. previous section

It enforces that the corresponding additive DPI has to satisfy:

$$\forall X_1, \dots, X_n, \llbracket \mathbf{f} \rrbracket(X_1, \dots, X_n) \geq \max(X_1, \dots, X_n)$$

Consequently, $\llbracket - \rrbracket$ is DPI then it is a quasi-interpretation and we obtain that the DPI synthesis problem is NP-hard over $\text{MaxPlus } \{\mathbb{K}\}$, $\mathbb{K} \in \{\mathbb{N}, \mathbb{Q}^+, \mathbb{Q}^+\}$. (2) is a direct consequence of (1) and Theorem 22 whereas (3) is a consequence of (1) and Theorem 23. \square

To conclude, we have found a better notion than the one of quasi-interpretation from an intensional point of view (i.e. in terms of algorithms) in order to get a sup-interpretation at equal cost from a synthesis point of view.

7. Runtime complexity

7.1. Runtime complexity functions as sup-interpretations

As previously stated, sup-interpretation is a tool that inherently deals with space consumption in an extensional way. Consequently, it is natural to link this notion with studies on time consumption of TRS. In an analogy with classical complexity theory, one could expect that a TRS running in polynomial time would lead the programmer to get a polynomial upper bound on the size of the computed value.

A good candidate for the notion of time complexity of a TRS is the notion of *runtime complexity* function, a function providing an upper bound on the length of the longest derivation with respect to the size of the initial term. Many studies have demonstrated that termination techniques can be used to study the runtime complexity of a given TRS. See [11, 12, 22, 36], among others. In this subsection, we show that, as expected, bounding the runtime complexity of a TRS, allows us to recover a sup-interpretation. For that purpose, we introduce usual definitions:

Definition 20. *The derivational length of a terminating term t with respect to a rewrite relation $\rightarrow_{\mathcal{R}}$ is defined by:*

$$\text{dl}(t, \rightarrow_{\mathcal{R}}) = \max\{n \in \mathbb{N} \mid \exists s, t \rightarrow_{\mathcal{R}}^n s\}$$

The runtime complexity function with respect to a rewrite relation $\rightarrow_{\mathcal{S}}$ on a set of terms T is defined by:

$$\text{rc}(n, T, \rightarrow_{\mathcal{S}}) = \max\{\text{dl}(t, \rightarrow_{\mathcal{S}}) \mid t \in T \text{ and } |t| \leq n\}$$

The runtime complexity function with respect to a TRS $\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle$ is defined by

$$\text{rc}_{\mathcal{R}}(n) = \text{rc}(n, T_b, \rightarrow_{\mathcal{R}})$$

where T_b is the set of basic terms of the shape $t = \mathbf{f}(v_1, \dots, v_n)$ with $\mathbf{f} \in \mathcal{D}$ and $v_1, \dots, v_n \in \text{Ter}(\mathcal{C})$.

We start to show an intermediate result that links the size of a term with respect to the length of its derivation. For that purpose, the size of a TRS $|\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle|$ is defined by $|\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle| = \sum_{l \rightarrow r \in \mathcal{R}} |l| + |r|$.

Lemma 5. *Given a TRS $\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle$, for every terms t, s and every $n \in \mathbb{N}$ such that $t \rightarrow_{\mathcal{R}}^n s$ we have:*

$$|s| \leq |t| \times |\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle|^n$$

Proof. By induction on the derivation length. If $n = 0$ then $t = s$ and we have $|t| \leq |t|$. Now suppose that it holds for a derivation of length $n - 1$ by induction hypothesis, that is $t \rightarrow_{\mathcal{R}}^{n-1} s$ and $|s| \leq |t| \times |\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle|^{n-1}$ and consider one more rewrite step $s \rightarrow_{\mathcal{R}}^1 u$. By definition of a rewrite step, there is a one-hole context $C[\diamond]$, a rule $l \rightarrow r \in \mathcal{R}$ and a substitution σ such that $s = C[l\sigma] \rightarrow_{\mathcal{R}} C[r\sigma] = u$. As a result, we obtain that:

$$\begin{aligned} |u| &= |C[r\sigma]| = |C[\diamond]| + |r\sigma| \\ &\leq |C[\diamond]| + |\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle| \times \max_{x \in \text{Var}(r)} |x\sigma| && \text{Since } |r| \leq |\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle| \\ &\leq |C[\diamond]| + |\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle| \times |l\sigma| && \text{Since } \text{Var}(r) \subseteq \text{Var}(l) \\ &\leq |s| \times |\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle| && \text{Since } |\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle| \geq 1 \end{aligned}$$

Combining both inequalities, we obtain $|u| \leq |s| \times |\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle| \leq |t| \times |\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle|^{n-1} \times |\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle|$ and so the result. \square

Now we relate runtime complexity to sup-interpretations:

Theorem 28. *Given a terminating TRS $\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle$, then the assignment θ defined by:*

- $\theta(\mathbf{c}) = 0$, if $\mathbf{c} \in \mathcal{C}$ is of arity 0
- $\theta(\mathbf{c})(X_1, \dots, X_n) = \sum_{i=1}^n X_i + 1$, if $\mathbf{c} \in \mathcal{C}$ is of arity $n > 0$
- $\theta(\mathbf{f})(X_1, \dots, X_n) = (\sum_{i=1}^n X_i + 1) \times |\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle|^{r_{\mathcal{C}\mathcal{R}}(\sum_{i=1}^n X_i + 1)}$, if $\mathbf{f} \in \mathcal{D}$

is a sup-interpretation.

Proof. Note that the assignment defined is clearly additive and monotonic. Consequently, we have to show that given a symbol $\mathbf{f} \in \mathcal{D}$ of arity n and values $v_1, \dots, v_n \in \text{Ter}(\mathcal{C})$, if $\mathbf{f}(v_1, \dots, v_n) \downarrow$ then $\theta(\mathbf{f}(v_1, \dots, v_n)) \geq \theta(\llbracket \mathbf{f} \rrbracket(v_1, \dots, v_n))$. Note that by definition of θ , $\forall v \in \text{Ter}(\mathcal{C})$, $\theta(v) = |v|$. Consider the reduction $\mathbf{f}(v_1, \dots, v_n) \rightarrow_{\mathcal{R}}^* \llbracket \mathbf{f} \rrbracket(v_1, \dots, v_n)$, we know that there exists $m \in \mathbb{N}$ such that $\mathbf{f}(v_1, \dots, v_n) \rightarrow_{\mathcal{R}}^m \llbracket \mathbf{f} \rrbracket(v_1, \dots, v_n)$ since the TRS is terminating. Consequently:

$$\begin{aligned} \theta(\llbracket \mathbf{f} \rrbracket(v_1, \dots, v_n)) &= |\llbracket \mathbf{f} \rrbracket(v_1, \dots, v_n)| \\ &\leq |\mathbf{f}(v_1, \dots, v_n)| \times |\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle|^m && \text{By Lemma 5} \\ &\leq \left(\sum_{i=1}^n |v_i| + 1 \right) \times |\langle \mathcal{D} \uplus \mathcal{C}, \mathcal{R} \rangle|^{r_{\mathcal{C}\mathcal{R}}(\sum_{i=1}^n |v_i| + 1)} && \text{By Def. 20} \\ &\leq \theta(\mathbf{f})(|v_1|, \dots, |v_n|) = \theta(\mathbf{f})(\theta(v_1), \dots, \theta(v_n)) \end{aligned}$$

and so the conclusion. \square

7.2. Sup-interpretations through termination techniques

Note that the bound provided in Theorem 28 is exponential in the length of a derivation. In general, it cannot be improved since it relies on the fact that most TRS do not compute terms of size polynomial in the reduction length because they make a strong use of variable duplication. However it can be improved by either syntactically restricting the set of considered TRS, considering for example, linear TRS, TRS that do not replicate their variables, or by semantically restricting the shape of the captured TRS wrt some termination technique fixed in advance. Additive polynomial interpretations described in Section 4 are an example of such a tool. Indeed they only capture programs computing polynomial size values. Note that the restriction lies in the additivity condition and no longer holds if we consider arbitrary interpretations.

We subsume the main termination techniques that can be used to infer sup-interpretations in the Figure 7.2.

Termination technique	SI upper bound	Synthesis problem
Polynomial interpretations	$O(2^{2^n})$	Undecidable
Additive Polynomial interpretations	$O(n^k), k \in \mathbb{N}$	Undecidable
Linear additive interpretations	$O(n)$	P
Restricted Matrix interpretations	$O(n^k), k \in \mathbb{N}$	NP
RPO	$O(f(n)), f \in \text{MR}$	NP-complete [†]
DP-based methods	$O(2^{f(n)}), f \in \mathcal{C}_{\text{DP}}$	NP

Figure 3:

In this Figure, the first column lists the termination tool under consideration. The second column provides an upper bound $O(g)$ on the sup-interpretations functions that can be computed with respect to the termination technique under consideration. More precisely, for a n -ary function symbol \mathbf{f} of a TRS whose termination has been shown using some fixed technique, it means that $\theta(\mathbf{f})(X_1, \dots, X_n) = h(\max_{1 \leq i \leq n}(X_i))$ is a sup-interpretation, for some function h such that $h = O(g)$. Finally, the last column corresponds to the complexity of the respective termination problem.

Now we briefly explain the results of Figure 7.2 line-by-line:

- For polynomial interpretations, the doubly exponential upper bound on the derivation length of a terminating TRS was shown by Hofbauer and

Lautemann in [11]. An important point to stress is that the obtained result for SI is finer than the one that could be expected by a naive application of Theorem 28: indeed the SI upper bound remains doubly exponential (and not a triple exponential). The undecidability result of the synthesis is demonstrated in Section 4 and was suggested by Lankford [8].

- If we consider additive polynomial interpretations, the synthesis remains undecidable however the sup-interpretation codomain is restricted to polynomials because the size of a value in $Ter(\mathcal{C})$ is exactly equal to its size. This result is due to Bonfante et al. [37].
- As a consequence, restriction to linear functions yields a linear upper bound computable in polynomial time using linear programming.
- For matrix interpretation techniques, [38] demonstrates that the runtime complexity is exponentially bounded in the height of a term. As a consequence, we obtain a double exponential upper bound when the height equals the size, by a naive application of Theorem 28. Note that this general framework can be restricted to $O(n^k)$, $k \in \mathbb{N}$ using polynomially bounded matrix interpretations of [39, 40] or context dependent interpretations [41] together with restrictions on the interpretation of constructor symbols, in the same spirit as additive polynomial interpretations. See also [42] for a generalization of matrix interpretations. The complexity of the synthesis is in NP because the algorithm that shows the termination of a TRS with matrix interpretations uses a SAT solver.
- The RPO termination technique gives an upper bound exponential in a function $f \in \text{MR}$, where MR stands for the set of multiple recursive functions. This upper bound relies on the lexicographic comparison which yields a multiple recursive derivation length as demonstrated by [36]. This bound can be improved to primitive recursive functions PR if we restrict to Multiset Path Ordering (MPO) as demonstrated by [43]. Note that we obtain the required upper bound on SI since both MR and PR are closed under exponentiation. The NP-completeness of RPO was demonstrated in [44]. († : Note that contrarily to previous mentioned techniques, this technique shows the existence of a SI but does not provide it explicitly.)
- For DP-based methods, Hirokawa and Moser have demonstrated in [45] that techniques combining Dependency Pairs and restricted Interpretations, named SLI for Strongly Linear Interpretations, yields $O(n^2)$ runtime complexity and, consequently, we obtain sup-interpretations in $O(2^{n^2})$ in this particular case. This technique can be generalized to arbitrarily large upper bounds on SI, depending on the base termination technique used. Consequently, the upper bound is $O(2^{f(n)})$ for some $f \in \mathcal{C}_{\text{DP}}$, where \mathcal{C}_{DP} is a set of runtime complexity functions induced by the DP-method under consideration. For example, the use of RPO as base technique would

give a SI in MR. Note that in this case, the exponential gap can also be withdrawn by putting extra restrictions on the termination system. The synthesis is also in NP because the verification is based on SAT solvers.

One of the main drawbacks in the use of runtime complexity in order to infer sup-interpretations is that we are restricted to terminating TRS and so, to total functions. From that point of view, it is important to stress that QI and DPI based techniques of Sections 5 and 6 allow for such a treatment because they do not imply termination even if they are based on polynomial interpretation methods. Consequently, they may allow the programmer to infer (polynomial) space upper bounds on the computed values (and also the intermediate values in the case of QI) even if the derivation length of the considered term is bounded by a function of high complexity.

8. Conclusion

In this paper, we have studied three methods (interpretations, quasi-interpretations and DP-interpretations) that define a sup-interpretation. Moreover, we have studied the complexity of the sup-interpretation synthesis problem on sets of polynomials including a max operator and we have shown that some termination techniques may allow the programmer to build sup-interpretations. One important issue that falls outside of the scope of this paper concerns the automation of the synthesis problem: in particular the search of efficient algorithms that could allow the programmer to obtain the sup-interpretation of programs (or TRS computing partial functions). Another important issue is to synthesize sup-interpretations through other techniques (type systems, ...). Moreover, we have restricted our discussion to monotonic sup-interpretations. An interesting challenge would be to obtain tighter upper bounds by considering non monotonic functions. It is a very difficult problem since all the techniques known to the author are based on monotonicity conditions. Finally, due to lack of space, we have not studied the synthesis problem over other paradigms. However we let the reader check that finding a sup-interpretation can always be turned into a constraint satisfaction, see [46] for example. Consequently, the complexity results presented in this paper have an impact that is not restricted to term rewriting.

- [1] J.-Y. Marion, R. Péchoux, Sup-interpretations, a semantic method for static analysis of program resources, ACM TOCL 10 (4) (2009) 1–31.
- [2] M. Bezem, J. Klop, R. de Vrijer, Term rewriting systems, Cambridge University Press, 2003.
- [3] F. Baader, T. Nipkow, Term Rewriting and All That, Cambridge University Press, 1998.
- [4] H. Rogers Jr, Theory of recursive functions and effective computability, MIT Press, 1987.

- [5] N. D. Jones, *Computability and complexity, from a programming perspective*, MIT press, 1997.
- [6] T. Arts, J. Giesl, Termination of term rewriting using dependency pairs, *Theor. Comput. Sci.* 236 (2000) 133–178.
- [7] G. Bonfante, J.-Y. Marion, J.-Y. Moyen, R. Péchoux, Synthesis of quasi-interpretations, LCC2005, LICS affiliated Workshop.
- [8] D. Lankford, On proving term rewriting systems are noetherian, Tech. rep. (1979).
- [9] Z. Manna, S. Ness, On the termination of Markov algorithms, in: *Third hawaii international conference on system science*, 1970, pp. 789–792.
- [10] E. Cichon, P. Lescanne, Polynomial interpretations and the complexity of algorithms, in: *CADE*, no. 607 in LNAI, 1992, pp. 139–147.
- [11] D. Hofbauer, C. Lautemann, Termination proofs and the length of derivations (preliminary version), in: *RTA*, Vol. 355 of LNCS, Springer, 1989, pp. 167–177.
- [12] G. Moser, A. Schnabl, Proving quadratic derivational complexities using context dependent interpretations, in: *RTA*, Vol. 5117 of LNCS, Springer, 2008, pp. 276–290.
- [13] J.-Y. Marion, J.-Y. Moyen, Efficient first order functional program interpreter with time bound certifications, in: *LPAR*, Vol. 1955 of LNCS, Springer, 2000, pp. 25–42.
- [14] G. Bonfante, F. Deloup, Complexity Invariance of Real Interpretations, in: *TAMC*, Vol. 6108 of LNCS, Springer, 2010, pp. 139–150.
- [15] G. Bonfante, J.-Y. Marion, J.-Y. Moyen, On lexicographic termination ordering with space bound certifications, in: *PSI*, Vol. 2244 of LNCS, Springer, 2001, pp. 482–493.
- [16] G. Bonfante, J.-Y. Marion, J.-Y. Moyen, Quasi-interpretations and small space bounds., in: *RTA*, Vol. 3467 of LNCS, Springer, 2005, pp. 150–164.
- [17] J.-Y. Marion, R. Péchoux, Characterizations of polynomial complexity classes with a better intensionality, in: *PPDP*, ACM, 2008, pp. 79–88.
- [18] R. Amadio, Synthesis of max-plus quasi-interpretations, *Fundamenta Informaticae* 65 (1–2).
- [19] C. Fuhs, J. Giesl, A. Middeldorp, P. Schneider-Kamp, R. Thiemann, H. Zankl, Sat solving for termination analysis with polynomial interpretations, in: *SAT*, LNCS, Springer, 2007, pp. 340–354.

- [20] C. Fuhs, J. Giesl, A. Middeldorp, P. Schneider-Kamp, R. Thiemann, H. Zankl, Maximal termination, in: RTA, Vol. 5117 of LNCS, Springer, 2008, pp. 110–125.
- [21] C. Borralleras, S. Lucas, A. Oliveras, E. Rodríguez-Carbonell, A. Rubio, Sat modulo linear arithmetic for solving polynomial constraints, *J. Autom. Reasoning* 48 (1) (2012) 107–131.
- [22] M. Avanzini, G. Moser, A. Schnabl, Automated implicit computational complexity analysis (system description), in: IJCAR, Vol. 5195 of LNCS, 2008, pp. 132–138.
- [23] G. Bonfante, J.-Y. Marion, R. Péchoux, Quasi-interpretation synthesis by decomposition, in: ICTAC, Vol. 4711 of LNCS, Springer, 2007, pp. 410–424.
- [24] S. Kleene, *Introduction to metamathematics*, Wolters-Noordhoff, 1988.
- [25] N. Dershowitz, A note on simplification orderings, *Information Processing Letters* 9 (5) (1979) 212–215.
- [26] S. Lucas, On the relative power of polynomials with real, rational, and integer coefficients in proofs of termination of rewriting, *Appl. Algebra Eng. Commun. Comput.* 17 (1) (2006) 49–73.
- [27] E. Contejean, C. Marché, A. Tomás, X. Urbain, Mechanically proving termination using polynomial interpretations, *J. Autom. Reasoning* 34 (4) (2005) 325–363.
- [28] S. Lucas, Practical use of polynomials over the reals in proofs of termination, in: PPDP, ACM, 2007, pp. 39–50.
- [29] F. Neurauter, A. Middeldorp, Polynomial interpretations over the reals do not subsume polynomial interpretations over the integers, in: RTA, Vol. 6 of LIPIcs, Springer, 2010, pp. 243–258.
- [30] Y. V. Matiyasevich, *Hilbert’s 10th Problem*, Foundations of Computing Series, MIT Press, 1993.
- [31] A. Tarski, *A Decision Method for Elementary Algebra and Geometry*, University of California Press, 1951.
- [32] G. E. Collins, Quantifier elimination for real closed fields by cylindrical algebraic decomposition, in: *Conference on Automata Theory and Formal Languages*, Vol. 33 of LNCS, 1975.
- [33] J. Heintz, M.-F. Roy, P. Solerno, Sur la complexité du principe de Tarski-Seidenberg, *Bulletin de la S.M.F.*, tome 118 (1990) 101–126.
- [34] G. Bonfante, J.-Y. Marion, J.-Y. Moyen, Quasi-interpretations a way to control resources, *Theor. Comput. Sci.* 412 (25) (2011) 2776–2796.

- [35] Y. Matiyasevich, M. Davis, Hilbert’s tenth problem, Vol. 94, MIT press, 1993.
- [36] A. Weiermann, Termination proofs for term rewriting systems by lexicographic path orderings imply multiply recursive derivation lengths, *Theor. Comput. Sci.* 139 (1&2) (1995) 355–362.
- [37] G. Bonfante, A. Cichon, J.-Y. Marion, H. Touzet, Algorithms with polynomial interpretation termination proof, *Journal of Functional Programming* 11 (1) (2001) 33–53.
- [38] J. Endrullis, J. Waldmann, H. Zantema, Matrix interpretations for proving termination of term rewriting, *J. Autom. Reasoning* 40 (2-3) (2008) 195–220.
- [39] J. Waldmann, Polynomially bounded matrix interpretations, in: *RTA*, Vol. 6 of *LIPICs*, 2010, pp. 357–372.
- [40] F. Neurauder, H. Zankl, A. Middeldorp, Revisiting matrix interpretations for polynomial derivational complexity of term rewriting, in: *LPAR (Yogyakarta)*, Vol. 6397 of *LNCS*, Springer, 2010, pp. 550–564.
- [41] G. Moser, A. Schnabl, J. Waldmann, Complexity analysis of term rewriting based on matrix and context dependent interpretations, in: *FSTTCS*, Vol. 2 of *LIPICs*, 2008, pp. 304–315.
- [42] A. Middeldorp, G. Moser, F. Neurauder, J. Waldmann, H. Zankl, Joint spectral radius theory for automated complexity analysis of rewrite systems, in: *CAI*, Vol. 6742 of *LNCS*, Springer, 2011, pp. 1–20.
- [43] D. Hofbauer, Termination proofs with multiset path orderings imply primitive recursive derivation lengths, *Theor. Comput. Sci.* 105 (1) (1992) 129–140.
- [44] M. S. Krishnamoorthy, P. Narendran, On recursive path ordering, *Theor. Comput. Sci.* 40 (2-3) (1985) 323–328.
- [45] N. Hirokawa, G. Moser, Automated complexity analysis based on the dependency pair method, in: *IJCAR*, Vol. 5195 of *LNCS*, Springer, 2008, pp. 364–379.
- [46] J.-Y. Marion, R. Pécoux, Analyzing the implicit computational complexity of object-oriented programs, in: *FSTTCS*, Vol. 2 of *LIPICs*, 2008, pp. 316–327.

3 Algebras and coalgebras in the light affine lambda calculus

Algebras and Coalgebras in the Light Affine Lambda Calculus

Marco Gaboardi, Romain Péchoux

► **To cite this version:**

Marco Gaboardi, Romain Péchoux. Algebras and Coalgebras in the Light Affine Lambda Calculus. The 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015), Aug 2015, Vancouver, Canada. hal-01112165

HAL Id: hal-01112165

<https://hal.inria.fr/hal-01112165>

Submitted on 9 Oct 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Algebras and Coalgebras in the Light Affine Lambda Calculus

Marco Gaboardi[‡] Romain Péchoux^{*}

[‡] Harvard University and University of Dundee ^{*} Université de Lorraine

Abstract

Algebra and coalgebra are widely used to model data types in functional programming languages and proof assistants. Their use permits to better structure the computations and also to enhance the expressivity of a language or of a proof system.

Interestingly, parametric polymorphism *à la* System F provides a way to encode algebras and coalgebras in strongly normalizing languages without loosing the good logical properties of the calculus. Even if these encodings are sometimes unsatisfying because they provide only limited forms of algebras and coalgebras, they give insights on the expressivity of System F in terms of functions that we can program in it.

With the goal of contributing to a better understanding of the expressivity of Implicit Computational Complexity systems, we study the problem of defining algebras and coalgebras in the Light Affine Lambda Calculus, a system characterizing the complexity class FPTIME. In this system, the principle of *stratification* limits the ways we can use parametric polymorphism, and in general the way we can write our programs.

We show here that while stratification poses some issues to the standard System F encodings, it still permits to encode some weak form of algebra and coalgebra. Using the algebra encoding one can define in the Light Affine Lambda Calculus the traditional inductive types. Unfortunately, the corresponding coalgebra encoding permits only a very limited form of coinductive data types. To extend this class we study an extension of the Light Affine Lambda Calculus by distributive laws for the modality \S . This extension has been discussed but not studied before.

1. Introduction

Algebras and coalgebras Data types shape the style we can use to write our programs, contributing in this way to determining the *expressivity* of a programming language. Algebras and coalgebras of a functor (see [26] for an extended introduction) are important tools coming from category theory that are useful to specify data types in a uniform way. This uniformity has been exploited in the design of functional programming languages, via the use of abstract data types. In this particular setting, algebraic types correspond usually to finite data types and coalgebraic ones correspond to infinite data types.

Despite the fact that coalgebras correspond to infinite data types, interestingly algebras and coalgebras can be also added to languages that are strongly normalizing by preserving the strong normalization property, as shown by Hagino [23]. Moreover, algebras and coalgebras can also be encoded by using parametric polymorphism in strongly normalizing languages as System F as shown by Wraith [41]. Preserving strong normalization corresponds to preserving the consistency property of the language. It is this last feature that allows the integration of algebras and coalgebras in proof assistants such as Coq and Agda, where they can be used to define inductive and coinductive data types, respectively.

Different notions of algebras and coalgebras can provide different forms of recursion and corecursion that can be used to program algorithms in different ways. Moreover, algebras and coalgebras

also provide some form of induction and coinduction that we can use to prove program properties. So, algebras and coalgebras are abstractions that are useful to compare in the abstract the expressivity of distinct languages.

Implicit Computational Complexity (ICC) ICC aims at characterizing complexity classes by means that are independent from the underlying machine model. A characterization of a complexity class C is traditionally determined by a system S obtained by restricting the class of proofs of a given logical system or the class of programs of a given programming language L . In order to characterize C , the system S needs to satisfy two properties: 1) the evaluation process of proofs or programs of S must lie within the given complexity class C —this ensures that S is *sound* with respect to C ; 2) any function (or decision problem) in C must be implementable by a proof or program in S —this ensures that S is *complete* with respect to C .

This approach for characterizing complexity classes where all the functions or problems of the given class C can be encoded by some proof or program in S is traditionally referred to as *extensionally complete*. On the other hand, an *intensionally complete* characterization requires that all the proofs or programs that can be evaluated within the complexity class C must lie in the restriction S . From a programmer’s perspective intensionally complete characterizations are certainly preferable to extensionally complete ones since they capture all the algorithms of the language L that lie in the class C . However, providing intensional characterizations of well-known and interesting complexity classes is in general problematic: for polynomial time the problem of providing an intensional characterization is Σ_2^0 -complete in the arithmetical hierarchy—and so undecidable—as proved by Hájek [24].

This contrast between extensional and intensional completeness has motivated researchers in ICC in the search of restrictions to logical systems and programming languages that are more and more expressive in terms of proofs or programs in L that they can fit. This is usually achieved in two ways: by weakening the restrictions, and by enriching the language with new programming constructs. See the survey by Hofmann [25] for more information.

Light Logics The Light Logics [21, 27] approach to ICC is based on the idea of providing characterizations of complexity classes by means of subsystems of Girard’s (second order) Linear Logic [20]. Proofs of second order linear logic can be seen through the proof-as-programs correspondence as terms of System F typed under a refined typing discipline using the contraction and weakening rules in a more principled way via the exponential modality $!$.

Following the light logic approach one can design type systems for the lambda calculus and its extensions where only programs that are in a particular computational complexity class can be assigned a type. This approach has been used to provide characterizations of several complexity classes like FPTIME [2, 5, 14, 39], PSPACE [16], LOGSPACE [38], NP [15, 32], P/Poly [33], etc.

A bird’s eye view on Light Affine Logic One of the most successful examples of light logic is certainly Light Affine Logic (LAL), the affine version of Light Linear Logic (LLL). Similarly to LLL,

LAL provides a characterization of the class FPTIME by limiting the way the modality $!$ is used in proofs and by introducing a new modality \S to compensate for some of these limitations. Roughly, the modality $!$ is used as a marker for objects that can be iterated, the modality \S is used as a marker of objects that are the result of an iteration and cannot be iterated anymore. The combined use of these two modalities provides a way to limit the iterations that one can write in proofs, and so the complexity of the systems.

More precisely, LAL enforces a design principle named *stratification* by adopting the following rules for modalities¹:

$$\frac{\Gamma \vdash \tau \quad \Gamma \subseteq \{\sigma\}}{!\Gamma \vdash !\tau} (!) \quad \frac{\Gamma, !\tau, !\tau \vdash \sigma}{\Gamma, !\tau \vdash \sigma} (C) \quad \frac{\Gamma, \Delta \vdash \tau}{!\Gamma, \S\Delta \vdash \S\tau} (\S)$$

The stratification is obtained by limiting the introduction of the two modalities to the two rules (!) and (\S), respectively. These rules can be seen as boxes that stratify the proofs. In other words, stratification corresponds roughly to ruling out the logical principles $!A \multimap A$ and $!A \multimap !!A$ but allowing the principles $!A \multimap \S A$. Enforcing stratification is not sufficient to characterize polynomial time—it provides a characterization of Elementary time [21]. For this reason, LAL further restrict the power of the modality $!$ by requiring that the environment Γ in the rule (!) has at most one assumption, this is the meaning of the premise $\Gamma \subseteq \{\sigma\}$. This requirement corresponds roughly to ruling out the logical principles $!A \otimes !B \multimap !(A \otimes B)$ and only allowing instead the restricted principle $!A \otimes !B \multimap \S(A \otimes B)$.

Despite their rather technical definitions, LLL and LAL provide natural and quite expressive characterizations of the class FPTIME . For this reason, their principles have been used to design a lambda calculus [39], a type system [5] and an extended language [6] for polynomial time computations. In these languages one can program several natural polynomial time algorithms over different data structures.

Our contribution In this work we study the definability of algebras and coalgebras in the Linear Affine Lambda Calculus (LALC), a term language for LAL, with the aim of better understanding the expressivity of LALC with respect to the definability of inductive and coinductive data structures, in particular with a focus on infinite data structures like streams.

Since LALC can be seen as a subsystem of System F, we study how to adapt the encoding of algebras and coalgebras in System F to the case of LALC. Not surprisingly, the standard System F encoding cannot be straightforwardly adapted to LALC because of the stratification principle. Indeed variable duplication in the terms enforces the modalities $!$ and \S to appear. These modalities propagate to the functor thus requiring types encoding initial algebras and final coalgebras that differ from the ones of the standard encoding. The initial algebra for the functor F can be encoded in LALC by terms of type

$$\forall X.!(F(X) \multimap X) \multimap \S X$$

The final coalgebra for the same functor F can instead be encoded by terms of type

$$\exists X.!(X \multimap F(X)) \otimes \S X$$

Initial algebras and final coalgebras definable in System F are only *weak*. In the case of LALC the two types above provide an even more restricted class of initial algebras and final coalgebras: intuitively, the ones that *behave well under \S* as a marker for iteration. These definitions will be made precise in Section 4 and Section 5, respectively. A further restriction comes from the fact that to obtain these classes of algebras and coalgebras we need

¹ We present here the rules of the logic in sequent calculus. The corresponding typing rules will then be presented in Section 3.

to consider only functors that behaves well with respect to the modality \S . More precisely, for initial algebras we need functors that *left-distribute* over \S , i.e. functors F such that $F(\S X) \multimap \S F(X)$. Conversely, for final algebras we need functors that *right-distribute* over \S , i.e. functors F such that $\S F(X) \multimap F(\S X)$.

Functors that left-distribute over \S are quite common in LALC and so we can define several standard inductive data types. Unfortunately, only few functors right-distribute over \S . In particular, we cannot encode standard coinductive data structures. The main reason is that the modality \S does not *distribute* with respect to the connectives tensor and plus. More precisely, in LALC we cannot derive the distribution² laws $\S(A \otimes B) \multimap \S A \otimes \S B$ and $\S(A \oplus B) \multimap \S A \oplus \S B$ for generic A and B . We overcome this situation by adding terms for these distributive laws to LALC. Thanks to this extensions we are able to write programs working on infinite streams of booleans (or of any finite data type) and other infinite data types.

Quite interestingly, Girard [19, §16.5.3] remarked that adding the principle $\S(A \oplus B) \multimap \S A \oplus \S B$ (“supposedly doing what one thinks”) to LAL would bring to the absurd situation where we can decide in linear time all the polynomial time problems. The informal argument is that this principle would allow us to extract the output bit of a decision problem without the need of computing it. A discussion on this argument has also been used by Baillet and Mazza [4] to explain one of the difference between LAL and their Linear Logic by Level. Here we show that adding the distributivity principle $\S(A \oplus B) \multimap \S A \oplus \S B$ with a computational counterpart in the term language does not bring to the absurd situation prospected by Girard. On the contrary, we show that the full evaluation of programs containing this distributivity principle requires a more complex reduction strategy than the dept-by-depth one traditionally used for LAL [21]. This is also reflected in the polynomial time soundness proof for LAL extended with distributions that we provide in Section 6. Let us stress that our argument is not necessarily in contradiction with Girard’s argument because the latter relies on the informal condition “supposedly doing what one thinks” and one can think to introduce the distributivity principles as an identity $\S(A \oplus B) = \S A \oplus \S B$ without computational content. In this case, the absurde situation would indeed arise.

2. Algebras and Coalgebras in System F

The starting point of our work is the encoding of weak initial F -algebras and weak final F -coalgebras in System F as described by Wraith [41] and Freyd [11] (see also Wadler [40]). Let us start by reviewing the definition of F -algebras and F -coalgebras.

Definition 1 (F -Algebra and F -Coalgebra). *Given a category \mathcal{C} and an endofunctor $F : \mathcal{C} \rightarrow \mathcal{C}$:*

- a F -algebra is pair (A, a) of an object $A \in \mathcal{C}$ together with a \mathcal{C} -morphism $a : F(A) \rightarrow A$,
- a F -coalgebra is pair (A, a) of an object $A \in \mathcal{C}$ together with a \mathcal{C} -morphism $a : A \rightarrow F(A)$.

Algebras and coalgebras provide the basic syntactic structure that is needed in order to define data types.

We can define two categories $\text{Alg-}F$ and $\text{Coalg-}F$ whose objects are F -algebras and F -coalgebras, respectively, and whose morphisms are defined as follows.

Definition 2. *A F -algebra homomorphism from the F -algebra (A, a) to the F -algebra (B, b) is a morphism $f : A \rightarrow B$ making*

² We use here the term “distribution” because we think to both modalities and type constructors as operations. In the literature, other people have preferred the term “commutation”.

the following diagram commute:

$$\begin{array}{ccc}
F(A) & \xrightarrow{F(f)} & F(B) \\
\downarrow a & & \downarrow b \\
A & \xrightarrow{f} & B
\end{array}$$

A F -coalgebra homomorphism from the F -coalgebra (A, a) to the F -coalgebra (B, b) is a morphism $f : A \rightarrow B$ making the following diagram commute:

$$\begin{array}{ccc}
A & \xrightarrow{f} & B \\
\downarrow a & & \downarrow b \\
F(A) & \xrightarrow{F(f)} & F(B)
\end{array}$$

To define the traditional inductive and coinductive data types we also need the notions of *initial algebras* and *final coalgebras*.

Definition 3 (Initial algebra and final coalgebra). A F -algebra (A, a) is initial if for each F -algebra (B, b) , there exists a unique F -algebra homomorphism $f : A \rightarrow B$. A F -coalgebra (A, a) is final if for each F -coalgebra (B, b) , there exists a unique F -coalgebra homomorphism $f : B \rightarrow A$.

If the uniqueness condition is not met then the F -algebra (resp. F -coalgebra) is only weakly initial (resp. weakly final).

An initial F -algebra is an initial object in the category $\text{Alg-}F$. Conversely, a final F -coalgebra is a terminal object in the category $\text{Coalg-}F$. In the definition above, the existence of a homomorphism provides a way to build objects by (co)iteration; this corresponds to have the ability to define by iteration elements in type fixpoints. Conversely, the uniqueness of such homomorphism provides a way to prove properties of these elements by (co)induction; this is something that type fixpoint does not necessarily provide.

Example 4. Consider the functor F defined by $F(X) = 1 + X$. The pair $(\mathbb{N}, [0, \text{succ}])$ consisting in the set of natural numbers \mathbb{N} together with the morphism $[0, \text{succ}] : 1 + \mathbb{N} \rightarrow \mathbb{N}$, defined as the coproduct of $0 : 1 \rightarrow \mathbb{N}$ and $\text{succ} : \mathbb{N} \rightarrow \mathbb{N}$, is an F -algebra.

Consider the endofunctor F over the category Set defined by $F(X) = A \times X$, for some set A . The pair $(A^\omega, \langle \text{head}, \text{tail} \rangle)$ where A^ω is the set of infinite lists over A and the morphism $\langle \text{head}, \text{tail} \rangle : A^\omega \rightarrow A \times A^\omega$ is defined by $\text{head} : A^\omega \rightarrow A$ and $\text{tail} : A^\omega \rightarrow A^\omega$, is a final F -coalgebra.

2.1 Encoding weak initial algebras and weak final coalgebras

We here assume some familiarity with System F and existential types (see [22] and [35]). A functor $F(X)$ is definable in System F if $F(X)$ is a type scheme mapping every type A to the type $F(A)$, and if there exists a term F mapping every term of type $A \rightarrow B$ to a term of type $F(A) \rightarrow F(B)$ and such that it preserves identity and composition. We say that a functor $F(X)$ is covariant if the variable X only appears in covariant positions.

It is a well known result that for any covariant functor $F(X)$ that is definable in System F we can define an algebra that is weakly initial and a coalgebra that is weakly final [26]. This corresponds to define the least and the greatest fixpoint of $F(X)$ as a type scheme.

Proposition 5 (Weak Initial Algebra). Let $F(X)$ be a covariant functor definable in System F and $T = \forall X.(F(X) \rightarrow X) \rightarrow X$. Consider the morphisms defined by:

$$\text{in}_T : F(T) \rightarrow T,$$

$$\text{in}_T = \lambda s : F(T).\Lambda X.\lambda k : F(X) \rightarrow X.k(F(\text{fold}_T X k) s),$$

$$\text{fold}_T : \forall X.(F(X) \rightarrow X) \rightarrow T \rightarrow X,$$

$$\text{fold}_T = \Lambda X.\lambda k : F(X) \rightarrow X.\lambda t : T.t X k.$$

Then, (T, in_T) is a weak initial F -algebra: for every F -algebra $(A, g : F(A) \rightarrow A)$ there is a F -homomorphism $h : T \rightarrow A$ defined as $h = \text{fold}_T A g$.

We will sometimes write T as $\mu X.F(X)$ when we want to stress the underlying functor F and the fact that T corresponds to the least fixpoint of F .

Proposition 6 (Weak Final Coalgebra). Let F be a covariant functor definable in System F and $T = \exists X.(X \rightarrow F(X)) \times X$. Consider the morphisms defined by:

$$\text{out}_T : T \rightarrow F(T),$$

$$\text{out}_T = \lambda t : T.\text{unpack } t \text{ as } (X, z) \text{ in}$$

$$\text{let } (k, x) = z \text{ in } F(\text{unfold}_T X k)(k x),$$

$$\text{unfold}_T : \forall X.(X \rightarrow F(X)) \rightarrow X \rightarrow T,$$

$$\text{unfold}_T = \Lambda X.\lambda k : X \rightarrow F(X).\lambda x : X.\text{pack } ((k, x), X) \text{ as } T.$$

Then, (T, out_T) is a weak final F -coalgebra: for every F -coalgebra $(A, g : A \rightarrow F(A))$ there is a F -homomorphism $h : A \rightarrow T$ defined as $h = \text{unfold}_T A g$.

Similarly to the case of F -algebras, we will write T as $\nu X.F(X)$ when we want to stress the underlying functor F and the fact that T corresponds to the greatest fixpoint of F .

Example 7. Let us consider a functor defined on types as $F(X) = 1 + X$ and on terms as:

$$\lambda f : X \rightarrow Y.\lambda x : 1 + X.\text{case } x \text{ of}$$

$$\{\text{inj}_0^{1+X}(z) \rightarrow \text{inj}_0^{1+Y}(), \text{inj}_1^{1+X}(z) \rightarrow \text{inj}_1^{1+Y}(f z)\}.$$

Let $\mathbb{N} = \mu X.F(X)$. Proposition 5 ensures that $(\mathbb{N}, \text{in}_{\mathbb{N}})$ is a weak initial algebra: the weak initial algebra of natural numbers. In particular, we can define $0 = \text{in}_{\mathbb{N}}(\text{inj}_0^{1+\mathbb{N}}())$, $n+1 = \text{in}_{\mathbb{N}}(\text{inj}_1^{1+\mathbb{N}}(n))$, and more in general the successor function as $\text{succ} = \lambda x.\text{in}_{\mathbb{N}}(\text{inj}_1^{1+\mathbb{N}}(x))$. We can use the fact that \mathbb{N} is a weak initial algebra to define an addition function. We just need to consider a term like the following (we omit some type for conciseness):

$$g = \lambda x : 1 + (\mathbb{N} \rightarrow \mathbb{N}).\text{case } x \text{ of}$$

$$\{\text{inj}_0(z) \rightarrow \lambda y : \mathbb{N}.y, \text{inj}_1(z) \rightarrow \lambda y : \mathbb{N}.\text{succ}(z y)\}.$$

Then, Proposition 5 ensures that we can define add as $\text{fold}_{\mathbb{N}}(\mathbb{N} \rightarrow \mathbb{N}) g$.

Example 8. Let us consider a functor defined on types as $F(X) = \mathbb{N} \times X$ and on terms as:

$$\lambda f : X \rightarrow Y.\lambda x : \mathbb{N} \times X.\text{let } \langle x_1, x_2 \rangle = x \text{ in } \langle x_1, f x_2 \rangle.$$

Let $\mathbb{N}^\omega = \nu X.F(X)$. Proposition 5 ensures that $(\mathbb{N}^\omega, \text{out}_{\mathbb{N}^\omega})$ is a weak final coalgebra: the weak final coalgebra of streams over natural numbers. We can define the usual operations on streams as $\text{head} = \lambda x : \mathbb{N}^\omega.\text{let } \langle x_1, x_2 \rangle = (\text{out}_{\mathbb{N}^\omega} x) \text{ in } x_1$, and $\text{tail} = \lambda x : \mathbb{N}^\omega.\text{let } \langle x_1, x_2 \rangle = (\text{out}_{\mathbb{N}^\omega} x) \text{ in } x_2$. We can use the fact that \mathbb{N}^ω is a weak final coalgebra to define streams. As an example we can define a constant stream of k s by using a function:

$$g = \lambda x : 1.\text{let } () = x \text{ in } \langle k, () \rangle.$$

$$\begin{aligned}
\tau, \sigma ::= & X \mid \mathbf{1} \mid !\tau \mid \S\tau \mid \tau \oplus \sigma \mid \tau \otimes \sigma \mid \tau \multimap \sigma \\
& \mid \forall X. \tau \mid \exists X. \tau \\
\mathbf{M}, \mathbf{N}, \mathbf{L} ::= & x \mid () \mid \lambda x : \tau. \mathbf{M} \mid \mathbf{M}\mathbf{N} \mid \Lambda X. \mathbf{M} \mid \mathbf{M}\tau \mid \hat{\mathbf{M}} \mid \hat{\S}\mathbf{M} \\
& \mid \text{let } \hat{\S}x : \tau = \mathbf{M} \text{ in } \mathbf{N} \mid \text{let } \hat{\mathbf{!}}x : \tau = \mathbf{M} \text{ in } \mathbf{N} \\
& \mid \langle \mathbf{M}, \mathbf{N} \rangle \mid \text{let } \langle x : \tau_1, y : \tau_2 \rangle = \mathbf{M} \text{ in } \mathbf{N} \\
& \mid \text{pack } (\mathbf{M}, \sigma) \text{ as } \tau \mid \text{unpack } \mathbf{M} \text{ as } (X, x) \text{ in } \mathbf{N} \\
& \mid \text{let } () = \mathbf{M} \text{ in } \mathbf{N} \mid \text{inj}_i^\tau(\mathbf{M}) \mid \\
& \mid \text{case } \mathbf{M} \text{ of } \{ \text{inj}_0^\tau(x) \rightarrow \mathbf{N} \mid \text{inj}_1^\tau(x) \rightarrow \mathbf{L} \}
\end{aligned}$$

Figure 1. LALC: grammar for types and terms.

Proposition 6 ensures that we can define $\text{const} = \text{unfold}_{\mathbb{N}^\omega} \mathbf{1} g()$. Similarly, we can define a function that extracts from a stream the elements in even position. This time we need a function:

$$g = \lambda x : \mathbb{N}^\omega. \langle \text{hd } x, \text{tl } (\text{tl } x) \rangle.$$

Proposition 6 ensures that we can define $\text{even} = \text{unfold}_{\mathbb{N}^\omega} \mathbb{N}^\omega g$.

3. The Light Affine Lambda Calculus

The Light Affine Lambda Calculus is the affine version of the Light Linear Lambda Calculus [39] and provide a concrete syntax for Intuitionistic Light Affine Logic [1].

3.1 The language

The syntax of the Light Affine Lambda Calculus (LALC) is inspired by the restrictions provided by Light Affine Logic. The focus of our work is on the expressivity of the calculus rather than on other properties, so to make our examples more clear we adopt an explicitly typed version of LALC. The types and the terms of LALC are presented in Figure 1. As basic type we consider only the multiplicative unit $\mathbf{1}$, while as type constructors we consider the linear implication \multimap , the tensor product \otimes , and the additive disjunction \oplus . Moreover, we have type variables X and type universal and existential quantifications: $\forall X. \tau$, and $\exists X. \tau$. We also have two modalities $!$, and \S . The connectives \otimes , \oplus and the existential quantification $\exists X. \tau$ can be defined by using only the linear implication \multimap , the modalities $!$, \S , and the universal quantification \forall but we prefer here to consider them as primitive. The reason behind this choice is that in the second part of the paper we will introduce explicit rules for distributing the \S modality over \otimes and \oplus , so it is natural to consider them as primitive.

Every type constructor comes equipped with a term constructor and a term destructor. Since we consider explicitly typed term we avoid confusions by denoting $\hat{\mathbf{!}}$ and $\hat{\S}$ the term level constructors for the modalities $!$ and \S , respectively. The semantics of LALC is defined in terms of the reduction relation \rightarrow described in Figure 2 where we use the notation $[\mathbf{M}/x]$ for the usual capture avoiding term substitution, the notation $[\tau/X]$ for the usual capture avoiding type substitution, and \dagger, \ddagger to denote the modalities $!$ or \S .

We have three kinds of reduction rules: the *exponential* rules describe the interaction of a constructor and a destructor for modalities, the *beta* rules describe instead the interaction of a constructor and a destructor for all the other types, the *commuting conversion* rules describe the interaction of different destructors. In Figure 2 we have omitted several commuting rules. The number of these rules is quite high and their behavior is standard. We consider only two such rules (com-1) and (com-2) as representative of this class.

3.2 Type system

A typing judgment is of the shape $\Gamma \vdash \mathbf{M} : \tau$, for some typing environment Γ (an environment assigning types to term variables),

some term \mathbf{M} and some type τ . The standard typing rules, inherited from Light Affine λ -calculus, are given in Figure 3(a) and additional rules for the extra constructs are given in Figure 3(b). As usual, this system uses the notion of discharged formulas, which are expressions of the form $[\tau]_{\dagger}$. Given a typing environment $\Gamma = x_1 : \tau_1, \dots, x_n : \tau_n$, $[\Gamma]_{\dagger}$ is a notation for the environment $x_1 : [\tau_1]_{\dagger}, \dots, x_n : [\tau_n]_{\dagger}$. Discharged formulas are not types, so they cannot be abstracted and we do not want them to appear in final judgments. They are just syntactic artifacts introduced by the rule $(!I)$ and $(\S I)$, used by the rule (C) , and eliminated by the rules $(!E)$ and $(\S E)$, respectively. These five rules implement in a natural deduction style the three sequent calculus LAL rules we discussed in the introduction and the *cut rule* on modalities. All the other rules are the linear versions of the standard System F rules. We assume a *multiplicative* management of contexts: when we write Γ, Δ we assume that the set of free variables in Γ and Δ are disjoint. The only rule that uses in part an *additive* management of context is the rule $(\oplus E)$ where we have a sharing of variables in the two branches of the case construction. We adopt here the same convention as in Light Linear Logic (LLL) [21] and we consider a *lazy* reduction that reduce redexes with variable bound in the two branches only when the argument is closed.

The polynomial soundness of LALC can be expressed in terms of the *depth* $d(\mathbf{M})$ of a term \mathbf{M} : the maximal number of nested $\hat{\mathbf{!}}$ or $\hat{\S}$ that can be found in any path of the term syntax tree. Moreover, we will call *depth* of a subterm \mathbf{N} of \mathbf{M} the number of $\hat{\mathbf{!}}$ and $\hat{\S}$ that one has to cross to reach the root of \mathbf{N} starting from the root of \mathbf{M} .

3.3 Properties of LALC

LALC provides a characterization of the FPTIME complexity class. However, it also enjoys standard properties of typed lambda calculi. In particular, it enjoys subject reduction.

Theorem 9 (Subject Reduction). *Let $\Gamma \vdash \mathbf{M} : \tau$ and $\mathbf{M} \rightarrow \mathbf{N}$, then $\Gamma \vdash \mathbf{N} : \tau$.*

In fact, LALC enjoys also a stronger version of subject reduction that ensures not only that types are preserved, but also that a reduction $\mathbf{M} \rightarrow \mathbf{N}$ corresponds to a rewriting of the type derivation of \mathbf{M} in the type derivation of \mathbf{N} .

Polynomial time soundness for LALC can be stated as follow:

Theorem 10 (Polynomial Time Soundness). *Consider a term $\Gamma \vdash \mathbf{M} : \tau$. Then, \mathbf{M} can be reduced to normal form by a Turing Machine working in time polynomial in $|\mathbf{M}|$ with exponent proportional to $d(\mathbf{M})$.*

The original proof of this theorem by Girard [21] as well as other subsequent proofs [2, 39] are based on three main observations about reductions in LALC:

1. reductions cannot increase the depth of a term,
2. a beta reduction at depth i decreases the size of the term at depth i and cannot increase the size of the term at other depths,
3. any sequence of exponential reduction at depth i can only square the size of the term at depth j greater than i .

These properties suggest a depth-by-depth reduction strategy whose length is polynomial in the size of the term and exponential in the depth.

For expressing the FPTIME completeness statement for LALC we need a data type \mathbb{B}^* for strings of Booleans that can be easily defined by a standard Church encoding.

Theorem 11 (FPTIME completeness). *For every polynomial time function $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ there exists a natural number n and a term $\mathbf{f} : \mathbb{B}^* \multimap \hat{\S}^n \mathbb{B}^*$ such that and \mathbf{f} represents f .*

$(\lambda x : \tau. M) N \rightarrow M[N/x]$	(beta- λ)
$\text{let } () = () \text{ in } M \rightarrow M$	(beta-1)
$(\Lambda X. M) \tau \rightarrow M[\tau/X]$	(beta- \forall)
$\text{case inj}_i^\tau(M) \text{ of } \{\text{inj}_0^\tau(x) \rightarrow N_0 \text{inj}_1^\tau(x) \rightarrow N_1\} \rightarrow N_i[M/x]$	(beta- \oplus)
$\text{let } (x : \tau, y : \sigma) = \langle N_0, N_1 \rangle \text{ in } M \rightarrow M[N_0/x, N_1/y]$	(beta- \otimes)
$\text{unpack } (\text{pack } (M, \sigma) \text{ as } \exists X. \tau) \text{ as } (X, x) \text{ in } N \rightarrow N[M/x, \sigma/X]$	(beta- \exists)
$\text{let } \hat{\S}x : \hat{\S}\tau = \hat{\S}N \text{ in } M \rightarrow M[N/x]$	(exp- $\hat{\S}$)
$\text{let } \hat{!}x : \hat{!}\tau = \hat{!}N \text{ in } M \rightarrow M[N/x]$	(exp- $\hat{!}$)
$(\text{let } \hat{\dagger}x : \hat{\dagger}\tau = N \text{ in } M) L \rightarrow \text{let } \hat{\dagger}x : \hat{\dagger}\tau = N \text{ in } (ML)$	(com-1)
$\text{let } \hat{\dagger}y : \hat{\dagger}\tau = (\text{let } \hat{\dagger}x : \hat{\dagger}\sigma = N \text{ in } L) \text{ in } M \rightarrow \text{let } \hat{\dagger}x : \hat{\dagger}\sigma = N \text{ in } (\text{let } \hat{\dagger}y : \hat{\dagger}\tau = L \text{ in } M)$	(com-2)

Figure 2. Light Affine Lambda Calculus reduction rules.

The proof of this statement requires to show that one can program in LALC all the polynomial expressions, that one can define data types for Turing Machine’s configurations, and that transitions between configurations are definable.

4. Algebras in LALC

4.1 Motivations and definition

We want now to adapt the encoding of algebras in System F to the case of LALC. The first thing that we need is to find a type that permits to express a weak initial F -algebra. One can consider the straightforward linear type $T = \forall X. (F(X) \multimap X) \multimap X$ but this is not enough for typing an analogous of the term in_T that contains a duplicated variable k . Consequently, modalities are required in the corresponding type as the duplication in the term in_T is needed for iteration. So, a more natural choice is instead to use the type:

$$T = \forall X. !(F(X) \multimap X) \multimap \S X. \quad (1)$$

Indeed, this type can be seen as the analogous of the one used for the standard Church natural numbers in LALC: $\forall X. !(X \multimap X) \multimap \S(X \multimap X)$. In this type, the modality $!$ is a marker for the duplication and the modality \S witness the iteration.

Using this type for assigning types in LALC to terms analogous to the one of Proposition 5 presents two problems. First, the modality \S in Equation 1—that witnesses iteration—propagates when one wants to build the F -homomorphism to another F -algebra. This implies that only a restricted form of weak initial algebras can be obtained. So, we are able to build the needed F -algebra homomorphisms only with F -algebra of the form $(\S B, g : F(\S B) \rightarrow \S B)$. There is a second problem: we need the functor F to *left-distribute* over \S ³. This corresponds to require the existence of a morphism:

$$L_F : F(\S X) \multimap \S F(X).$$

This technical requirement comes from the fact that the modality \S propagates due to the iteration, but also from the uniformity imposed by the polymorphic encoding. This uniformity corresponds to require that the algebra $(\S B, g)$ target of the F -algebra homomorphism comes from an underlying F -algebra (B, f) via the functoriality of \S and the left-distributivity L_F of F .

By considering the two requirements, we obtain the following definition.

Definition 12. *Given a functor F , we say that an F -algebra (A, a) is weakly-initial under \S if for every F -algebra of the form (B, f) there exists an F -algebra $(\S B, g)$ and an F -algebra homomor-*

phism $h : A \rightarrow \S B$ making the following diagram commute:

$$\begin{array}{ccc}
 F(A) & \xrightarrow{F(h)} & F(\S B) \\
 \downarrow a & & \downarrow g \\
 A & \xrightarrow{h} & \S B
 \end{array}
 \begin{array}{l}
 \\
 \swarrow L_F \\
 \searrow \S f
 \end{array}
 \begin{array}{c}
 \\
 \\
 \S F(B)
 \end{array}$$

That is, we require the existence of an F -algebra homomorphism only for F -algebra of the form $(\S B, g)$ that comes from an underlying F -algebra (B, f) via the functoriality of \S and the left-distributivity L_F of F . With these two restrictions in mind we can now formulate an analogous of Proposition 5.

Theorem 13. *Let F be a functor definable in LALC that left-distributes over \S , and let $T = \forall X. !(F(X) \multimap X) \multimap \S X$. Consider the morphisms defined by:*

$$\begin{aligned}
 \text{in}_T &: F(T) \multimap T, \\
 \text{in}_T &= \lambda s : F(T). \Lambda X. \lambda k : !(F(X) \multimap X). \\
 &\quad \text{let } \hat{!}y : !(F(X) \multimap X) = k \text{ in} \\
 &\quad \text{let } \hat{\S}z : \S F(X) = L_F(F(\text{fold}_T X \hat{!}y)s) \text{ in } \hat{\S}(y z),
 \end{aligned}$$

$$\begin{aligned}
 \text{fold}_T &: \forall X. !(F(X) \multimap X) \multimap T \multimap \S X, \\
 \text{fold}_T &= \Lambda X. \lambda k : !(F(X) \multimap X). \lambda t : T. t X k.
 \end{aligned}$$

Then, (T, in_T) is a weakly-initial F -algebra under \S : for every F -algebra $(B, f : F(B) \multimap B)$ we have an F -algebra $(\S B, g : F(\S B) \rightarrow \S B)$ and an F -algebra homomorphism $h : T \rightarrow \S B$ defined as $h = \text{fold}_T B \hat{!}f$.

The situation described by Theorem 13 corresponds to saying that for every F -algebra (B, f) the following diagram commutes:

$$\begin{array}{ccc}
 F(T) & \xrightarrow{F(\text{fold}_T B \hat{!}f)} & F(\S B) \\
 \downarrow \text{in}_T & & \downarrow g \\
 T & \xrightarrow{\text{fold}_T B \hat{!}f} & \S B
 \end{array}
 \begin{array}{l}
 \\
 \swarrow L_F \\
 \searrow \S f
 \end{array}
 \begin{array}{c}
 \\
 \\
 \S F(B)
 \end{array}$$

³ We call this property “distribute” instead of “commute” in order to highlight the distinction with the standard commuting rules (com-n).

$$\begin{array}{c}
\frac{}{\mathbf{x} : \tau \vdash \mathbf{x} : \tau} (Ax) \quad \frac{\Gamma \vdash \mathbf{M} : \tau}{\Gamma, \Delta \vdash \mathbf{M} : \tau} (W) \quad \frac{\Gamma, \mathbf{x} : [\tau]!, \mathbf{y} : [\tau]! \vdash \mathbf{M} : \sigma}{\Gamma, \mathbf{z} : [\tau]! \vdash \mathbf{M}[\mathbf{z}/\mathbf{x}, \mathbf{z}/\mathbf{y}] : \sigma} (C) \quad \frac{\Gamma, \mathbf{x} : \tau \vdash \mathbf{M} : \sigma}{\Gamma \vdash \lambda \mathbf{x} : \tau. \mathbf{M} : \tau \multimap \sigma} (\multimap I) \\
\frac{\Gamma \vdash \mathbf{M} : \tau \multimap \sigma \quad \Delta \vdash \mathbf{N} : \tau}{\Gamma, \Delta \vdash \mathbf{M}\mathbf{N} : \sigma} (\multimap E) \quad \frac{\Gamma \vdash \mathbf{N} : !\tau \quad \Delta, \mathbf{x} : [\tau]! \vdash \mathbf{M} : \sigma}{\Gamma, \Delta \vdash \mathbf{let} \hat{\mathbf{x}} : !\tau = \mathbf{N} \mathbf{in} \mathbf{M} : \sigma} (!E) \quad \frac{\Gamma \vdash \mathbf{N} : \S\tau \quad \Delta, \mathbf{x} : [\tau]_{\S} \vdash \mathbf{M} : \sigma}{\Gamma, \Delta \vdash \mathbf{let} \hat{\mathbf{x}} : \S\tau = \mathbf{N} \mathbf{in} \mathbf{M} : \sigma} (\S E) \\
\frac{\Gamma \vdash \mathbf{M} : \tau \quad \Gamma \subseteq \{\mathbf{x} : \sigma\}}{[\Gamma]!, \hat{\mathbf{M}} : !\tau} (!I) \quad \frac{\Gamma, \Delta \vdash \mathbf{M} : \tau}{[\Gamma]!, [\Delta]_{\S} \vdash \hat{\mathbf{M}} : \S\tau} (\S I) \quad \frac{\Gamma \vdash \mathbf{M} : \tau \quad X \notin \text{FTV}(\Gamma)}{\Gamma \vdash \Lambda X. \mathbf{M} : \forall X. \tau} (\forall I) \quad \frac{\Gamma \vdash \mathbf{M} : \forall X. \tau}{\Gamma \vdash \mathbf{M}\sigma : \tau[\sigma/X]} (\forall E) \\
\text{(a) standard rules}
\end{array}$$

$$\begin{array}{c}
\frac{\Gamma \vdash \mathbf{M} : \tau \quad \Delta \vdash \mathbf{N} : \sigma}{\Gamma, \Delta \vdash \langle \mathbf{M}, \mathbf{N} \rangle : \tau \otimes \sigma} (\otimes I) \quad \frac{\Gamma \vdash \mathbf{M} : \tau \otimes \sigma \quad \Delta, \mathbf{x} : \tau, \mathbf{y} : \sigma \vdash \mathbf{N} : \tau'}{\Gamma, \Delta \vdash \mathbf{let} \langle \mathbf{x} : \tau, \mathbf{y} : \sigma \rangle = \mathbf{M} \mathbf{in} \mathbf{N} : \tau'} (\otimes E) \quad \frac{}{\Gamma \vdash () : \mathbf{1}} (!I) \\
\frac{\Gamma \vdash \mathbf{M} : \mathbf{1} \quad \Delta \vdash \mathbf{N} : \tau}{\Gamma, \Delta \vdash \mathbf{let} () = \mathbf{M} \mathbf{in} \mathbf{N} : \tau} (!E) \quad \frac{\Gamma \vdash \mathbf{M} : \tau_i}{\Gamma \vdash \mathbf{inj}_i^{\tau_0 \oplus \tau_1}(\mathbf{M}) : \tau_0 \oplus \tau_1} (\oplus I) \quad \frac{\Gamma \vdash \mathbf{M} : \tau[\sigma/X]}{\Gamma \vdash \mathbf{pack}(\mathbf{M}, \sigma) \mathbf{as} \exists X. \tau : \exists X. \tau} (\exists I) \\
\frac{\Gamma \vdash \mathbf{M} : \tau_0 \oplus \tau_1 \quad \Delta, \mathbf{x} : \tau_0 \vdash \mathbf{N}_0 : \tau \quad \Delta, \mathbf{x} : \tau_1 \vdash \mathbf{N}_1 : \tau}{\Gamma, \Delta \vdash \mathbf{case} \mathbf{M} \mathbf{of} \{\mathbf{inj}_0^{\tau_0 \oplus \tau_1}(\mathbf{x}) \rightarrow \mathbf{N}_0 | \mathbf{inj}_1^{\tau_0 \oplus \tau_1}(\mathbf{x}) \rightarrow \mathbf{N}_1\} : \tau} (\oplus E) \quad \frac{\Gamma \vdash \mathbf{M} : \exists X. \tau \quad \Delta, \mathbf{x} : \tau \vdash \mathbf{N} : \sigma}{\Gamma, \Delta \vdash \mathbf{unpack} \mathbf{M} \mathbf{as} (X, \mathbf{x}) \mathbf{in} \mathbf{N} : \sigma} (\exists E) \\
\text{(b) rules for additional constructions}
\end{array}$$

Figure 3. Typing rules for the Light Affine Lambda Calculus.

This diagram provides a way to encode the least fixpoint of types, similarly to what we have for System F, and so to define standard data types.

Notice that with respect to initial algebras we have now relaxed both the uniqueness and the existence property.

4.2 Algebra examples

Before providing examples of algebras, we want to characterize a large class of functors that left-distribute.

Lemma 14. *All the functors built using the following signature left-distribute over \S :*

$$F(X) ::= \mathbf{1} \mid X \mid A \mid \S F(X) \mid F(X) \oplus F(X) \mid F(X) \otimes F(X),$$

provided that A is a closed type for which it exists a closed term of type $A \multimap !A$ or type $A \multimap \S A$.

Proof. By induction on $F(X)$. The full proof is in the supplementary material. \square

Thanks to the above lemma we can give a notion of weakly-initial F -algebra under \S to several standard examples.

Example 15. *Consider the functor $F(X) = \mathbf{1} \oplus X$. This is the linear analogous of the functor considered in Example 7, definable by the same term (in the types annotation, implication is replaced by a linear arrow and $+$ is replaced by \oplus). By Lemma 14, we have that F left-distribute over \S , and so by Theorem 13 we have that $(\mathbb{N}, \mathbf{in}_{\mathbb{N}})$ is a weakly-initial F -algebra under \S , where by abuse of notation we again use \mathbb{N} to denote $\mu X. F(X)$. Similarly to what we did in Example 7, we can define natural numbers as inhabitants of this type. Noticing that to the term g defined there we can also give the type $F(\mathbb{N} \multimap \mathbb{N}) \multimap (\mathbb{N} \multimap \mathbb{N})$, we have that $\mathbf{add} = \mathbf{fold}_{\mathbb{N}}(\mathbb{N} \multimap \mathbb{N}) \hat{!} g$ has type $\mathbb{N} \multimap \S(\mathbb{N} \multimap \mathbb{N})$.*

Example 16. *Consider the functor $F_n(X) = \mathbf{1} \oplus (\mathbb{B}_n \otimes X)$ where \mathbb{B}_n is as a finite type with n states. The functor $F_n(X)$ is definable*

by the term:

$$\lambda \mathbf{f} : X \multimap Y. \lambda \mathbf{x} : F_n(X). \mathbf{case} \mathbf{x} \mathbf{of}$$

$$\{\mathbf{inj}_0(\mathbf{z}) \rightarrow \mathbf{inj}_0(()),$$

$$\mathbf{inj}_1(\mathbf{z}) \rightarrow \mathbf{let} \langle \mathbf{z}_1 : \mathbb{B}_n, \mathbf{z}_2 : X \rangle = \mathbf{z} \mathbf{in} \mathbf{inj}_1(\langle \mathbf{z}_1, \mathbf{f} \mathbf{z}_2 \rangle)\},$$

where we omit the superscripts of the \mathbf{inj}_i constructs for readability. It is easy to verify that by Lemma 14, we have that F_n left-distribute over \S . So, if we define $\mathbb{B}_n^* = \mu X. F_n(X)$, by Theorem 13 we have that $(\mathbb{B}_n^*, \mathbf{in}_{\mathbb{B}_n^*})$ is a weakly-initial F -algebra under \S . In the particular case where $n = 2$, let $\mathbb{B}_2 = \mathbf{1} \oplus \mathbf{1}$ be the type for booleans. The type \mathbb{B}_2^* is inhabited by finite boolean strings: $\mathbf{nil} = \mathbf{in}_{\mathbb{B}_2^*}(\mathbf{inj}_0(()))$, $\mathbf{cons} = \lambda \mathbf{h} : \mathbb{B}_2. \lambda \mathbf{t} : \mathbb{B}_2^*. \mathbf{in}_{\mathbb{B}_2^*}(\mathbf{inj}_1(\langle \mathbf{h}, \mathbf{t} \rangle))$.

We can define a map function on boolean strings using the function:

$$g = \lambda \mathbf{f} : \mathbb{B}_2 \multimap \mathbb{B}_2. \lambda \mathbf{x} : F_2(\mathbb{B}_2^*). \mathbf{case} \mathbf{x} \mathbf{of}$$

$$\{\mathbf{inj}_0(\mathbf{z}) \rightarrow \mathbf{in}_{\mathbb{B}_2^*}(\mathbf{inj}_0(())),$$

$$\mathbf{inj}_1(\mathbf{z}) \rightarrow \mathbf{let} \langle \mathbf{z}_1 : \mathbb{B}_2, \mathbf{z}_2 : \mathbb{B}_2^* \rangle = \mathbf{z} \mathbf{in} \mathbf{in}_{\mathbb{B}_2^*}(\mathbf{inj}_1(\langle \mathbf{f} \mathbf{z}_1, \mathbf{z}_2 \rangle)).$$

Noticing that for a variable $\mathbf{f} : \mathbb{B}_2 \multimap \mathbb{B}_2$, to the term $g \mathbf{f}$ we can give the type $F(\mathbb{B}_2^*) \multimap \mathbb{B}_2^*$, we have that $\mathbf{map} \mathbf{f} = \mathbf{fold}_{\mathbb{B}_2^*} \mathbb{B}_2^* \hat{!} (g \mathbf{f})$ has type $\mathbb{B}_2^* \multimap \S \mathbb{B}_2^*$.

The next section will show an extensive example in programming with these data structures.

4.3 Polynomial time completeness of LALC using algebras

As a sanity check, we want to use the new encoding of algebras to prove the FPTIME completeness of LALC. This follows the same line as the standard proof from Asperti and Roversi [2] or the one from Baillot et al. [6]. The idea is to show that we can use algebras to encode polynomial expressions—that can be used as clocks for iterations—and Turing Machines and their transitions.

We have seen how to define natural numbers as inhabitants of the type $\mathbb{N} = \mu X. \mathbf{1} \oplus X$. It is important to stress that all the inhabitants of \mathbb{N} can be typed with a fixed number of $(!I)$ and $(\S I)$ rules—this corresponds to having terms with constant *depth* as defined in Definition 23—this is quite standard but still important to stress in order to ensure a sound characterization of PTIME,

see [28]. We want now to show that we can encode polynomial expressions and that we can use them as clocks of iterators. Let us start with the latter. Thanks to Theorem 13 and the fact that given two terms of type $\mathbf{1} \multimap A$ and $A \multimap A$ we can build a term of type $\mathbf{1} \oplus A \multimap A$ combining them, we can define an iteration scheme parametrized by the type A over natural numbers as:

$$\text{iter} : \mathbb{N} \multimap (\mathbf{1} \multimap A) \multimap (A \multimap A) \multimap \S A.$$

This term has the property that for every n , given a base b a step s it produces the n -th iteration of s over b . More precisely:

$$\text{iter } n \ b \ s \rightarrow^* \S(s^n \ b).$$

As an example, we can make explicit the base and the iteration step for addition $\text{add} : \mathbb{N} \multimap \S(\mathbb{N} \multimap \mathbb{N})$:

$$\begin{aligned} \text{b}_{\text{add}} &= \lambda z : \mathbf{1}. \text{let } () = z \text{ in } \lambda y. y : \mathbf{1} \multimap (\mathbb{N} \multimap \mathbb{N}), \\ \text{s}_{\text{add}} &= \lambda z. \lambda y. \text{succ}(zy) : (\mathbb{N} \multimap \mathbb{N}) \multimap (\mathbb{N} \multimap \mathbb{N}). \end{aligned}$$

The type of add is not entirely satisfying, however we can define a coercion function:

$$\begin{aligned} \text{coer} &: (\mathbb{N} \multimap \S(\mathbb{N} \multimap \mathbb{N})) \multimap \mathbb{N} \multimap \S\mathbb{N} \multimap \S\mathbb{N}, \\ \text{coer} &= \lambda f. \lambda n. \lambda m. \text{let } \S u = m \text{ in } (\text{let } \S z = f \ n \text{ in } \S(z \ u)). \end{aligned}$$

Moreover, we have a coercion function:

$$\begin{aligned} \text{coer}' &: \mathbb{N} \multimap \S\mathbb{N}, \\ \text{coer}' &= \lambda n. \text{iter } n \ \underline{0} \ \text{succ}. \end{aligned}$$

Thanks to these coercion functions we can change the type of addition:

$$\begin{aligned} \text{add}_c &: \mathbb{N} \multimap \mathbb{N} \multimap \S\mathbb{N}, \\ \text{add}_c &= \lambda n : \mathbb{N}. \lambda m : \mathbb{N}. \text{coer } \text{add } n \ (\text{coer}' \ m). \end{aligned}$$

This is the same type that we can assign to addition in LLL. While this type is good for adding together several elements, in order to define multiplication it is also convenient to give to addition the following type:

$$\begin{aligned} \text{add}_{\S} &: \S\mathbb{N} \multimap \S^2\mathbb{N} \multimap \S^2\mathbb{N}, \\ \text{add}_{\S} &= \lambda n : \S\mathbb{N}. \lambda m : \S^2\mathbb{N}. \text{let } \S u = n \\ &\quad \text{in let } \S w = m \text{ in } \S(\text{coer } \text{add } u \ w). \end{aligned}$$

We can now define the base step and the iteration step for multiplication $\text{mul} : \mathbb{N} \multimap \S(\mathbb{N} \multimap \S^2\mathbb{N})$:

$$\begin{aligned} \text{b}_{\text{mul}} &= \lambda z : \mathbf{1}. \text{let } () = z \text{ in} \\ &\quad \lambda y : !\mathbb{N}. \text{let } !v = y \text{ in } \S(\text{coer}' \ v) : \mathbf{1} \multimap (!\mathbb{N} \multimap \S^2\mathbb{N}), \\ \text{s}_{\text{mul}} &= \lambda g : !\mathbb{N} \multimap \S^2\mathbb{N}. \lambda y : !\mathbb{N}. \text{let } !z = y \text{ in} \\ &\quad (\text{add}_{\S} \ \S z \ (g \ !z)) : (!\mathbb{N} \multimap \S^2\mathbb{N}) \multimap (!\mathbb{N} \multimap \S^2\mathbb{N}). \end{aligned}$$

By using another coercions similar to coer and coer' we can assign to multiplication a type as: $\text{mul}_c : \mathbb{N} \multimap !\mathbb{N} \multimap \S^3\mathbb{N}$. By using addition and multiplication we can prove the following.

Lemma 17. *For any polynomial $p[x]$ in the variable x there exists an integer n and a term $\lambda x. \underline{p}$ of type $\mathbb{N} \multimap \S^n\mathbb{N}$ representing $p[x]$.*

Likewise Asperti and Roversi [2] the above lemma can be also improved by expressing the number n of \S in term of the degree of the polynomial. However, in our case we would have some extra \S given by the extra \S that we have in the type of mul_c .

Now we need to encode Turing Machines \mathcal{M} . We can encode a configuration of \mathcal{M} with n states by a type $\mathbb{M}_n = \mathbb{B}_3^* \otimes \mathbb{B}_3^* \otimes \mathbb{B}_n$ where \mathbb{B}_3^* is the type of strings over a three symbols alphabet, and \mathbb{B}_n is a finite type of length n . The first \mathbb{B}_3^* represents the left part of the tape while the second one represents the right part of the tape, starting from the scanned symbol. The type \mathbb{B}_n represents the state. The transition function δ between configurations can be defined by case analysis: we can represent δ by a term $\text{delta} : \mathbb{M}_n \multimap \mathbb{M}_n$.

So, we can use the iteration scheme to iterate transitions starting from an initial configuration. We then have the following:

Theorem 18 (FPTIME completeness). *For every polynomial time function $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ there exists a natural number n and a term $\mathfrak{f} : \mathbb{B}_3^* \multimap \S^n\mathbb{B}_3^*$ such that and \mathfrak{f} represents f .*

The proof uses the type of configurations as described above and is similar to the one presented by Asperti and Roversi [2] or the one presented by Baillot et al. [6].

5. Coalgebras in LALC

Trying to adapt the encoding of final coalgebras we hit unsurprisingly the same problem we had for the encoding of initial algebra. Knowing now the receipt we can consider the type:

$$T = \exists X. !(X \multimap F(X)) \otimes \S X. \quad (2)$$

By duality, we are able to build F -coalgebra homomorphisms only with coalgebras of the shape $(\S B, g : \S B \multimap F(\S B))$ and we will require the functor F to *right-distribute* over \S . The latter corresponds to require the existence of a morphism:

$$R_F : \S F(X) \multimap F(\S X).$$

These requirements come once again from the fact that the modality \S propagates and from the polymorphic encoding. We have the following dual of Definition 12.

Definition 19. *Given a functor F , we say that an F -coalgebra (A, a) is weakly-final under \S if for every F -coalgebra of the form (B, f) there exists an F -coalgebra $(\S B, g)$ and an F -coalgebra homomorphism $h : \S B \rightarrow A$ making the following diagram commute:*

$$\begin{array}{ccc} A & \xleftarrow{h} & \S B \\ \downarrow a & & \downarrow g \\ F(A) & \xleftarrow{F(h)} & F(\S B) \end{array} \quad \begin{array}{c} \searrow \S f \\ \nearrow R_F \end{array}$$

That is, we require the existence of an F -coalgebra homomorphism only for F -coalgebra of the form $(\S B, g)$ that comes from an underlying F -coalgebra (B, f) via the functoriality of \S and the right-distributivity R_F of F .

We can now formulate an analogous of Proposition 6.

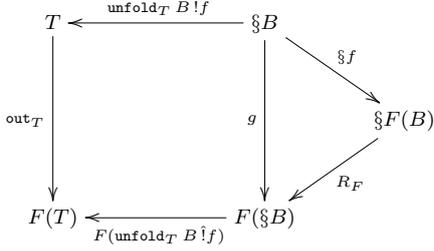
Theorem 20. *Let F be a functor definable in LALC that right-distribute over \S , and let $T = \exists X. !(X \multimap F(X)) \otimes \S X$. Consider the morphisms defined by:*

$$\begin{aligned} \text{out}_T &: T \multimap F(T), \\ \text{out}_T &= \lambda t : T. \text{unpack } t \text{ as } (X, z) \text{ in} \\ &\quad \text{let } \langle k : !(X \multimap F(X)), x : \S X \rangle = z \text{ in} \\ &\quad \text{let } !u = k \text{ in} \\ &\quad \text{let } \S v = x \text{ in } F(\text{unfold}_T \ X \ !u) R_F(\S(u \ v)), \\ \text{unfold}_T &: \forall X. !(X \multimap F(X)) \multimap \S X \multimap T, \\ \text{unfold}_T &= \lambda X. \lambda k : !(X \multimap F(X)). \lambda x : \S X. \text{pack } (\langle k, x \rangle, X) \text{ as } T. \end{aligned}$$

Then, (T, out_T) is a weakly-final F -coalgebra under \S : for every F -coalgebra $(B, f : B \multimap F(B))$ we have an F -coalgebra $(\S B, g : \S B \multimap F(\S B))$ and an F -coalgebra homomorphism $h : \S B \rightarrow T$ defined as $h = \text{unfold}_T \ B \ !f$.

The situation described by Theorem 13 corresponds to saying that for every F -coalgebra (B, f) the following diagram com-

mutates:



This diagram provides a way to encode the greatest fixpoint of a type schema, similarly to what we have for System F, and so to define standard data types.

Once again, with respect to final coalgebras we have now relaxed both the uniqueness and the existence property.

5.1 Lack of examples

We want now to give an analogous of Lemma 14 describing the functors that right-distribute. Unfortunately, we are able to prove only the following weak lemma.

Lemma 21. *All the functors built using the following signature right-distribute over §:*

$$F(X) ::= \mathbf{1} \mid X \mid A \mid \S F(X),$$

provided that A is a closed type for which it exists a closed term of type $\S A \multimap A$.

Clearly, this lemma is too weak for defining interesting coinductive examples. Indeed, even a simple functor such as $F(X) = \mathbf{1} \oplus X$ (the type for the natural numbers extended with an infinite object) is not right-distribute. By proof search it is easy to verify that we cannot derive a closed judgment for $\S(\mathbf{1} \oplus X) \multimap \mathbf{1} \oplus \S X$. Similarly, we cannot derive a closed judgment for the type $\S(A \otimes X) \multimap A \otimes \S X$, or the type $\S(A \otimes A \otimes X) \multimap A \otimes A \otimes \S X$. So, we cannot encode infinite lists and infinite trees using Theorem 20.

The root of this problem lies in the fact that in LALC, as in Light Linear Logic, the modality § does not distribute on the right with \otimes and \oplus . That is, we cannot prove in general $\S(A \otimes B) \multimap \S A \otimes \S B$ or $\S(A \oplus B) \multimap \S A \oplus \S B$. Without these distributive rules it seems hard to program any interesting coinductive data in LALC. For this reason, the next step is to add these rules to our language.

6. LALC with Distributions

We want now to add to LALC the distributive principles we discussed in the previous section. The calculus we obtain by adding these principles inherits the polynomial time completeness of LAL. However, we need to do some work in order to show that it preserves the *polynomial time soundness*. Indeed the usual proof technique based on the depth by depth reduction (see [39]) cannot be applied to this calculus as distributions at depth i may create new redexes at depth $i - 1$.

6.1 Extending LALC with distributions

We extend the grammar of Figure 1 with distributions, constructs that allow the reduction to distribute a § constructor over a $\text{inj}_i^\tau(-)$ or $\langle -, - \rangle$ constructor:

$M, N, L ::= \dots$

$$\mid \text{dist } \S(x : \tau_1, y : \tau_2) = M \text{ as } z = \langle \S x, \S y \rangle \text{ in } N$$

$$\mid \text{dist } \S \text{inj}_i^{\tau \oplus \tau'}(x) = M \text{ as } z = \text{inj}_i^{\S \tau \oplus \S \tau'}(\S x) \text{ in } N.$$

We extend the reduction relation \rightarrow from Figure 2 with two new distributive rules given in Figure 4 and as usual we denote by

\rightarrow^* its reflexive, transitive and contextual closure. Similarly, we add the two typing rules for distributions given in Figure 5. In the following, we will sometimes need to consider a term M with a specific type derivation Π for it, we will then use the notation $\Pi \triangleright \Gamma \vdash M : \tau$ for some environment Γ and type τ .

The most interesting logical properties of LAL, such as subject reduction, are preserved by this extension.

Theorem 22 (Subject reduction). *Let Π be a type derivation of the shape $\Pi \triangleright \Gamma \vdash M : \tau$. If $M \rightarrow N$, then there exists a type derivation $\Pi' \triangleright \Gamma \vdash N : \tau$.*

We have already discussed informally the depth of LAL terms. Here we introduce this concept more formally.

Definition 23 (depth). *Given a term M , the depth of M , noted $d(M)$, is the maximal number of nested \dagger constructs in a path of the syntax tree of M . Given a term M and one of its subterms N , N is at depth i in M if it is in the scope of i nested \dagger in M . Given a derivation Π , the depth of Π , noted $d(\Pi)$, is the maximal number of introduction rules for \dagger ($\dagger I$ rules) in a branch of the derivation Π . A rule in a given type derivation is at depth i if it is preceded by i $\dagger I$ rules in a branch of Π .*

Notice that the depth of a term coincides with the depth of its typing derivations. In other words, if $\Pi \triangleright \Gamma \vdash M : \tau$ then $d(M) = d(\Pi)$. In the following we will be interested in reductions that occur at a particular depth i , in such cases we will use the notation \rightarrow_i . Similarly to LAL, in the extended calculus the depth of a term cannot increase in the reduction.

Lemma 24 (Depth preservation). *Given two terms M and N , if $M \rightarrow^* N$ then $d(N) \leq d(M)$.*

Proof. By a case analysis of the reduction rules. See the supplementary material for more details. \square

6.2 Depth-by-depth reduction is not enough

We want now show that the depth-by-depth reduction is not enough to evaluate terms to normal form. Let us first introduce the notion of potential redex.

Definition 25 (Potential redexes at depth i). *A potential redex in M is a subterm whose outermost construct is either a destructor or a distribution. Given a type derivation $\Pi \triangleright \Gamma \vdash M : \tau$, we denote by $\mathcal{E}_i(\Pi)$ the number of elimination rules, $(d\oplus)$ and $(d\otimes)$ rules that are at depth i in Π . An actual redex in M is a potential redex that can be reduced at the outermost level by applying either a beta rule, an exponential rule or a distribution rule. A stuck redex in M is a potential redex that is not an actual redex.*

Notice that $\mathcal{E}_i(\Pi)$ is exactly equal to the number of potential redexes at depth i in M . Indeed, each elimination or distribution rule corresponds to the introduction of exactly one destructor or distribution construct in the typed term and conversely.

Fact 26 (From potential redex of depth i to actual redex of depth $i - 1$). *A reduction $M \rightarrow_i N$ at depth i can turn a stuck redex at depth $i - 1$ in an actual redex at depth $i - 1$.*

We show that this fact holds by providing an illustrating example. Consider a term M of depth i having the following subterm M_1 occurring at depth $i - 1$:

$$\text{dist } \S(x : \tau, y : \sigma) = \S(\lambda u : \tau'. \langle u, v \rangle) w \text{ as } z = \langle \S x, \S y \rangle \text{ in } M_2.$$

This subterm is a stuck redex as it is a potential redex (indeed a distribution) that is not an actual redex as the term $(\lambda u : \tau'. \langle u, v \rangle) w$ is not a constructor. Indeed, this term needs to be evaluated first for M_1 to become an actual redex. Moreover this term is of depth i in M

$$\text{dist } \mathbb{S}\langle x : \tau, y : \sigma \rangle = \mathbb{S}\langle M_1, M_2 \rangle \text{ as } z = \langle \mathbb{S}x, \mathbb{S}y \rangle \text{ in } N \rightarrow N[\langle \mathbb{S}M_1, \mathbb{S}M_2 \rangle / z] \quad (\text{dis-1})$$

$$\text{dist } \mathbb{S}\text{inj}_1^{\tau \oplus \tau'}(x) = \mathbb{S}\text{inj}_1^{\tau \oplus \tau'}(M) \text{ as } z = \text{inj}_1^{\mathbb{S}\tau \oplus \mathbb{S}\tau'}(\mathbb{S}x) \text{ in } N \rightarrow N[\text{inj}_1^{\mathbb{S}\tau \oplus \mathbb{S}\tau'}(\mathbb{S}M) / z] \quad (\text{dis-2})$$

Figure 4. Distributive reduction rules.

$$\frac{\Gamma \vdash M : \mathbb{S}(\tau \otimes \sigma) \quad \Delta, z : \mathbb{S}\tau \otimes \mathbb{S}\sigma \vdash N : \tau'}{\Gamma, \Delta \vdash \text{dist } \mathbb{S}\langle x : \tau, y : \sigma \rangle = M \text{ as } z = \langle \mathbb{S}x, \mathbb{S}y \rangle \text{ in } N : \tau'} \quad (\text{d}\otimes)$$

$$\frac{\Gamma \vdash M : \mathbb{S}(\tau \oplus \sigma) \quad \Delta, z : \mathbb{S}\tau \oplus \mathbb{S}\sigma \vdash N : \tau'}{\Gamma, \Delta \vdash \text{dist } \mathbb{S}\text{inj}_1^{\tau \oplus \sigma}(x) = M \text{ as } z = \text{inj}_1^{\mathbb{S}\tau \oplus \mathbb{S}\sigma}(\mathbb{S}x) \text{ in } N : \tau'} \quad (\text{d}\oplus)$$

Figure 5. Distributive typing rules for LALC

as it is located under an extra \mathbb{S} construct in M_1 . Consequently, we have $M \rightarrow_i N$ where N is the term obtained from M by substituting M_1 with the term N_1 below:

$$\text{dist } \mathbb{S}\langle x : \tau, y : \sigma \rangle = \mathbb{S}\langle w, v \rangle \text{ as } z = \langle \mathbb{S}x, \mathbb{S}y \rangle \text{ in } M_2.$$

N_1 is a potential redex of depth $i - 1$ in N (no enclosing \dagger -box has been changed wrt to the initial term) and is not a stuck redex since it reduces to $M_2[\langle \mathbb{S}w, \mathbb{S}v \rangle / z]$.

This example shows that we cannot use the depth-by-depth strategy to prove the polynomial time soundness of LALC extended with distributions. Indeed, we need a strategy that can round trip on the depth. In the following section we will prove the polynomial time soundness by using a global argument that provides an upper bound on the number and size of subterms generated in any reduction. This will help us in showing that each reduction has a polynomially bounded length.

6.3 Soundness of the extension

In this section, we show that the well-typed terms of our language can be reduced in polynomial time by a Turing Machine.

Preliminary notations We write $M \rightarrow_c N$ (resp. $M \rightarrow_{nc} N$) to stress that $M \rightarrow N$ and that the reduction uses (resp. does not use) a commutation rule (com-n). Similarly, we write $M \rightarrow_{k,i} N$, $k \in \{c, nc\}$, to stress that $M \rightarrow_k N$ with a reduction at depth i .

While the subject reduction only claims the existence of a type derivation Π' , the proof gives us a concrete way to build Π' starting from Π . Thanks to this we can lift our reasoning from terms to type derivations. Indeed every reduction in terms corresponds to some transformation on the type derivation. In general we will write $\Sigma : \Pi \triangleright M \mathcal{R}^* \Pi' \triangleright N$, with $\mathcal{R} \in \{\rightarrow, \rightarrow_i, \rightarrow_c, \rightarrow_{nc}, \rightarrow_{c,i}, \rightarrow_{nc,i}\}$, when we want to explicitly give a name Σ to the reduction and the corresponding type derivation Π' obtained by reducing M to N wrt the reflexive and transitive closure of \mathcal{R} starting with the type derivation Π . This will be useful when discussing about the structural rules—contraction and weakening—that do not have a corresponding syntactic construct in the language. So, they can only be seen in the typing derivation.

Length and size We also need to clarify the notions of length of a derivation and size of a term (or of a typing derivation). The reduction name Σ is useful when we want to deal with reduction length: $|\Sigma|$ will denote the length of the reduction (i.e. number of applications of \mathcal{R}), while $|\Sigma|^c$ (respectively $|\Sigma|^{nc}$) will denote the number of commutation rules (resp. rules that are not commutation rules) in Σ . Trivially, $|\Sigma| = |\Sigma|^c + |\Sigma|^{nc}$ as each reduction rule is either a commutation or not. The size of a term M is the number of subterms at any depth. The size of a typing derivation Π is the

total number of rules in it. Straightforwardly, the size of a term is bounded by the size of its typing derivations:

Lemma 27 (Size). *If $\Pi \triangleright \Gamma \vdash M : \tau$ then $|M| \leq |\Pi|$.*

Polynomial size reducts The key property that we will use to prove the soundness with a global argument is that the size of each intermediate type derivation obtained in a reduction is bounded polynomially by the size of the initial type derivation. To express this fact we need to refer to specific parts of a given term or derivation that are at a particular depth.

Definition 28 (size at depth i). *Given a term M we denote by $|M|_i$ the number of subterms of M that are at depth i in M . Trivially, the following equality holds $|M| = \sum_{i=0}^{d(M)} |M|_i$. In the same way, we denote by $|\Pi|_i$ the number of rules that are at depth i and the equality $|\Pi| = \sum_{i=0}^{d(\Pi)} |\Pi|_i$ also holds.*

We also need to count the contraction rules at each depth.

Definition 29 (contractions at depth i). *Given a type derivation $\Pi \triangleright \Gamma \vdash M : \tau$, we denote by $\mathcal{C}_i(\Pi)$ the number of contraction rules that are at depth i in Π .*

The notion of *potential* for a type derivation Π is introduced in order to bound the size of typing derivations in a reduction.

Definition 30. *Given a typing derivation $\Pi \triangleright \Gamma \vdash M : \tau$ we define its weight at depth i , denoted $\mathcal{W}_i(\Pi)$, by:*

$$\mathcal{W}_0(\Pi) = \mathcal{C}_0(\Pi)$$

$$\mathcal{W}_{i+1}(\Pi) = \prod_{j=0}^i (\mathcal{C}_j(\Pi) + 1)^{2^{i-j}} \cdot \mathcal{C}_{i+1}(\Pi)$$

The potential of Π , denoted $\mathcal{P}(\Pi)$ is defined as:

$$\mathcal{P}(\Pi) = \sum_{i=0}^{d(\Pi)} (\mathcal{W}_{i-1}(\Pi) + 1) \cdot |\Pi|_i$$

with the convention that $\mathcal{W}_{-1}(\Pi) = 0$.

We can now formulate the key property of the potential.

Lemma 31 (Polynomial size). *Consider a reduction $\Sigma : \Pi \triangleright M \rightarrow^* \Pi' \triangleright N$. Then, there is a polynomial $\mathcal{P}(\Pi)$ in $|\Pi|$ such that the following hold:*

- $|\Pi'| \leq \mathcal{P}(\Pi)$,
- $\mathcal{P}(\Pi) = \mathcal{O}(|\Pi|^{2^{d(\Pi)}+2})$.

Proof. By induction on the depth, using an upper bound on the maximal number of contraction rules that might exist at each depth

in a redex. The full proof are available in the supplementary material. \square

As a corollary, we obtain an upper bound on the number of potential redexes at each depth:

Corollary 32 (Polynomial number of potential redexes at any depth). *Consider a reduction $\Sigma : \Pi \triangleright \mathbb{M} \rightarrow^* \Pi' \triangleright \mathbb{N}$. Then, for any $i \geq 0$, we have: $\mathcal{E}_i(\Pi') \leq \mathcal{P}(\Pi)$.*

Proof. By Lemma 31 as $\mathcal{E}_i(\Pi') \leq |\Pi'| \leq \mathcal{P}(\Pi)$. \square

Polynomial length reductions Now we are ready to show the reduction length of a typed term is polynomially bounded by the size of its typing derivation. We proceed in two steps. First we show that the number of non commuting reduction steps is bounded polynomially in the size (and exponentially in the depth - that is fixed) of a term using a lexicographic decrease on the number of potential redexes (and the fact that they are bounded by the potential by Corollary 32). In a second step, we show that the number of commuting reduction steps is bounded polynomially in the size using a rewriting argument based on the structure of such rules.

Lemma 33. *Consider a reduction $\Sigma : \Pi \triangleright \mathbb{M} \rightarrow_{k,i} \Pi' \triangleright \mathbb{N}$, with $k \in \{c, nc\}$.*

(i) *For each $j < i$, we have $\mathcal{E}_j(\Pi') = \mathcal{E}_j(\Pi)$.*

(ii) *Moreover,*

1. *if $k = c$ then we have: for each $j \geq i$, $\mathcal{E}_j(\Pi') = \mathcal{E}_j(\Pi)$;*
2. *if $k = nc$ then we have: $\mathcal{E}_i(\Pi') < \mathcal{E}_i(\Pi)$.*

The above lemma tells us that a commutative reduction does not change potential redexes at all while non commutative reductions at depth i preserve potential redexes at depth strictly smaller than i and decrease by one the number of potential redexes at depth i . Of course, in this latter case, the number of potential redexes at depth strictly higher than i may increase.

Definition 34 (Strength). *Given a type derivation $\Pi \triangleright \Gamma \vdash \mathbb{M} : \tau$, the strength of Π , noted $s(\Pi)$, is a $d(\Pi) + 1$ tuple defined by:*

$$s(\Pi) = \langle \mathcal{E}_0(\Pi), \mathcal{E}_1(\Pi), \dots, \mathcal{E}_{d(\Pi)}(\Pi) \rangle.$$

Corollary 35. *Consider a reduction $\Sigma : \Pi \triangleright \mathbb{M} \rightarrow_{k,i} \Pi' \triangleright \mathbb{N}$, with $k \in \{c, nc\}$.*

1. *if $k = c$ then we have $s(\Pi) =_{lex} s(\Pi')$;*
2. *if $k = nc$ then we have: $s(\Pi) >_{lex} s(\Pi')$.*

where $>_{lex}$ is the lexicographic strict order induced by $>$ on tuples.

Now we take benefits of the above Corollary together with the previous upper bound on size of reduced proof in order to infer an upper bound on the length of non-commutative reductions.

Lemma 36. *Consider a reduction $\Sigma : \Pi \triangleright \mathbb{M} \rightarrow^* \Pi' \triangleright \mathbb{N}$. Then, we have:*

$$|\Sigma|^{nc} \leq (\mathcal{P}(\Pi))^{d(\Pi)+1}.$$

Proof. Let $t(\Pi) = \langle \mathcal{P}(\Pi), \dots, \mathcal{P}(\Pi) \rangle$ be a tuple with $d(\Pi) + 1$ times elements. By several applications of Corollary 35 and Corollary 32, for any reduction of the shape $\Pi_1 \triangleright \mathbb{M}_1 \rightarrow_c^* \Pi_2 \triangleright \mathbb{M}_2 \rightarrow_{nc} \Pi_3 \triangleright \mathbb{M}_3 \rightarrow_c^* \Pi_4 \triangleright \mathbb{M}_4$, the following holds:

$$\begin{array}{ccccccc} t(\Pi) & & t(\Pi) & & t(\Pi) & & t(\Pi) \\ \geq & & \geq & & \geq & & \geq \\ s(\Pi_1) & =_{lex} & s(\Pi_2) & >_{lex} & s(\Pi_3) & =_{lex} & s(\Pi_4), \end{array}$$

where \geq is the pointwise partial order induced by \geq on tuples.

Consequently, there can be no more than $(\mathcal{P}(\Pi))^{d(\Pi)+1}$ strict lexicographic decreases in a reduction. This provides us a bound on the number of reduction rules that are not commutation rules. \square

It is worth noticing that the above lemma diverges from the classical soundness proof for LAL. In particular, we combine a global argument given by the potential $\mathcal{P}(\Pi)$ of the type derivation Π with the lexicographic order that provides a local argument. In this way we have an argument that is independent from the strategy. While this approach has the consequence that the bound we provide is looser than the usual one, the difference is just in a small exponential constant that leaves the bound polynomial once the depth is fixed. A tighter bound—similar to the usual one—can be obtained by considering instead a specific reduction strategy where lower depth redexes are reduced with higher priority.

As usual, the length of reductions only involving commutation is bounded quadratically in the size of the initial type derivation using a term rewriting argument.

Lemma 37. *For each reduction $\Sigma : \Pi \triangleright \mathbb{M} \rightarrow_c^* \Pi' \triangleright \mathbb{N}$, we have $|\Sigma| \leq |\Pi|^2$.*

Now we are ready to show that any reduction has a polynomially bounded length:

Lemma 38 (Polynomially bounded reduction length). *Consider a reduction $\sigma : \Pi \triangleright \mathbb{M} \rightarrow^* \Pi' \triangleright \mathbb{N}$. Then, we have:*

$$|\Sigma| \leq \mathcal{P}(\Pi)^{d(\Pi)+1} \cdot (\mathcal{P}(\Pi)^2 + 1).$$

Proof. The inequality follows by combining Lemma 37, Corollary 31 and Lemma 36. See the supplementary material for the full proof. \square

FTime soundness We can now show the soundness properties of our type system:

Theorem 39 (Polynomial Time Soundness). *Consider a type derivation $\Pi \triangleright \Gamma \vdash \mathbb{M} : \tau$. Then, \mathbb{M} can be reduced to normal form by a Turing Machine working in time polynomial in $|\mathbb{M}|$ with exponent proportional to $d(\mathbb{M})$.*

Proof. By definition of depth, the equality $d(\Pi) = d(\mathbb{M})$ holds. Moreover, for any typable term \mathbb{M} , there is at least one normal type derivation Π (with no superfluous contraction rule and weakening rule). For such a type derivation, $|\Pi| = \mathcal{O}(|\mathbb{M}|)$ holds. By Lemma 31, the potential of a derivation is bounded by a polynomial in the size of Π with exponent proportional to $d(\Pi)$. By Lemma 38, each typable term \mathbb{M} has at most a polynomially bounded number of reductions in $|\mathbb{M}|$. By Corollary 31 the size of every intermediate term in the reduction is bounded polynomially in $|\mathbb{M}|$. As the reduction of LAL can be easily implemented by a Turing Machine in quadratic time (see [39]), the conclusion follows. \square

7. Examples

We can now improve Lemma 21 and obtain the following analogous of Lemma 14.

Lemma 40. *All the functors built using the following signature righ-distribute over \S :*

$F(X) ::= \mathbf{1} \mid X \mid A \mid \S F(X) \mid F(X) \otimes F(X) \mid F(X) \oplus F(X)$,
provided that A is a closed type for which it exists a closed term of type $\S A \multimap A$.

Proof. By induction on $F(X)$. \square

Similarly to Example 8 we would like to consider the case of streams of natural numbers. Unfortunately, Lemma 40 is not enough to show that the functor $F(X) = \mathbb{N} \otimes X$ distributes to the right. The problem is that we do not have a coercion $\S \mathbb{N} \multimap \mathbb{N}$, but only the converse one. Nevertheless, we can consider streams of booleans (or more in general of every finite type $\mathbf{1} \oplus \dots \oplus \mathbf{1}$).

Let us consider the functor defined on types as $F(X) = \mathbb{B}_2 \otimes X$ and on terms as:

$$\lambda f : X \multimap Y. \lambda x : \mathbb{B}_2 \otimes X. \text{let } \langle x_1, x_2 \rangle = x \text{ in } \langle x_1, f x_2 \rangle.$$

This functor right-distributes by Lemma 40. Let $\mathbb{B}_2^\omega = \nu X. F(X)$. Theorem 20 ensures that $(\mathbb{B}_2^\omega, \text{out}_{\mathbb{B}_2^\omega})$ is a weak final coalgebra. Let us use this property to define a constant stream of 1 (as a boolean). Similarly to what we did in Example 8 this can be defined by considering the function:

$$g = \lambda x : \mathbf{1}. \text{let } () = x \text{ in } \langle \mathbf{1}, () \rangle.$$

By Theorem 20 we can then define $\text{ones} = \text{unfold } \mathbf{1} \hat{!} g \hat{\S} ()$. Similarly to what happens in System F we can define the usual operations on streams as:

$$\text{head} = \lambda x : \mathbb{B}_2^\omega. \text{let } \langle x_1, x_2 \rangle = (\text{out}_{\mathbb{B}_2^\omega} x) \text{ in } x_1,$$

$$\text{tail} = \lambda x : \mathbb{B}_2^\omega. \text{let } \langle x_1, x_2 \rangle = (\text{out}_{\mathbb{B}_2^\omega} x) \text{ in } x_2.$$

Unfortunately, using these operations is often inconvenient in presence of linearity, and it is more convenient to use directly the coalgebra structure provided by $\text{out}_{\mathbb{B}_2^\omega}$. Consider for example the operation that extracts from a stream of booleans the elements in even position – we have seen a similar operation encoded in System F in Example 8. We can define this operation by using the function:

$$g = \lambda x : \mathbb{B}_2^\omega. \text{let } \langle x_1, x_2 \rangle = (\text{out}_{\mathbb{B}_2^\omega} x) \text{ in} \\ \text{let } \langle x_{21}, x_{22} \rangle = (\text{out}_{\mathbb{B}_2^\omega} x_2) \text{ in } \langle x_1, x_{22} \rangle.$$

This function has type $g : \mathbb{B}_2^\omega \multimap \mathbb{B}_2 \otimes \mathbb{B}_2^\omega$. So, by Theorem 20 $\text{even} = \text{unfold}_{\mathbb{B}_2^\omega} \mathbb{B}_2^\omega \hat{!} g$. Another interesting example is a function that merges two streams. This can be defined by the term:

$$g = \lambda x : \mathbb{B}_2^\omega \otimes \mathbb{B}_2^\omega. \text{let } \langle x_1, x_2 \rangle = x \text{ in} \\ \text{let } \langle x_{11}, x_{12} \rangle = (\text{out}_{\mathbb{B}_2^\omega} x_1) \text{ in } \langle x_{11}, \langle x_2, x_{12} \rangle \rangle.$$

This function has type $g : \mathbb{B}_2^\omega \otimes \mathbb{B}_2^\omega \multimap \mathbb{B}_2 \otimes (\mathbb{B}_2^\omega \otimes \mathbb{B}_2^\omega)$. So, by Theorem 20 again, $\text{merge} = \text{unfold}_{\mathbb{B}_2^\omega} (\mathbb{B}_2^\omega \otimes \mathbb{B}_2^\omega) \hat{!} g$.

We can combine algebra examples and coalgebra ones. For instance, we can write a standard inductive function take that for a given n returns the first n elements of a stream as a string. This can be obtained by the function:

$$g = \lambda x : \mathbf{1} \oplus (\mathbb{B}_2^\omega \multimap \mathbb{B}_2^*) . \text{case } x \text{ of} \\ \{ \text{inj}_0(z) \rightarrow \lambda y : \mathbb{B}_2^\omega. \text{nil} \mid \text{inj}_1(z) \rightarrow \\ \lambda y : \mathbb{B}_2^\omega. \text{let } \langle y_1, y_2 \rangle = (\text{out}_{\mathbb{B}_2^\omega} y) \text{ in } \text{cons}(y_1, z y_2) \}.$$

This function has type $g : \mathbf{1} \oplus (\mathbb{B}_2^\omega \multimap \mathbb{B}_2^*) \multimap (\mathbb{B}_2^\omega \multimap \mathbb{B}_2^*)$. So, by Theorem 13, $\text{take} = \text{fold}_{\mathbb{B}_2^\omega} (\mathbb{B}_2^\omega \multimap \mathbb{B}_2^*) \hat{!} g$.

Even if we cannot define a stream of the standard inductive natural numbers, we can have a stream of extended natural numbers. Let us define the latter first. Consider the functor $F(X) = \mathbf{1} \oplus X$. By Lemma 40, we have that F right-distributes over $\hat{\S}$, and so by Theorem 20 we have that $(\bar{\mathbb{N}}, \text{out}_{\bar{\mathbb{N}}})$ is a weakly-final F-coalgebra under $\hat{\S}$, where $\bar{\mathbb{N}}$ denotes $\nu X. F(X)$. The inhabitants of the type $\bar{\mathbb{N}}$ correspond to the natural numbers extended with a limit element ∞ . We can think about $\text{out}_{\bar{\mathbb{N}}}$ as a predecessor function mapping 0 to $()$, n to $n - 1$ and ∞ to ∞ . We can define the addition of two extended natural numbers by considering the term:

$$g = \lambda x : \bar{\mathbb{N}} \otimes \bar{\mathbb{N}}. \text{let } \langle x_1, x_2 \rangle = x \text{ in } \left(\text{case } (\text{out}_{\bar{\mathbb{N}}} x_1) \text{ of} \right. \\ \left. \left\{ \text{inj}_0(z) \rightarrow \text{case } (\text{out}_{\bar{\mathbb{N}}} x_2) \text{ of } \{ \text{inj}_0(z') \rightarrow \text{inj}_0(()) \right. \right. \\ \left. \left. \quad \left. \mid \text{inj}_1(z') \rightarrow \text{inj}_1(\langle z, z' \rangle) \right\} \right. \right. \\ \left. \mid \text{inj}_1(z) \rightarrow \text{inj}_1(\langle z, x_2 \rangle) \right\} \Bigg\}.$$

This function has type $g : \bar{\mathbb{N}} \otimes \bar{\mathbb{N}} \multimap \mathbf{1} \otimes (\bar{\mathbb{N}} \otimes \bar{\mathbb{N}})$. So, by Theorem 20 $\text{add} = \text{unfold}_{\bar{\mathbb{N}}} (\bar{\mathbb{N}} \otimes \bar{\mathbb{N}}) \hat{!} g$. For the extended natural numbers, we have a term $\text{coer}_{\bar{\mathbb{N}}} : \hat{\S} \bar{\mathbb{N}} \multimap \bar{\mathbb{N}}$, this is given by Theorem 20 as $\text{coer}_{\bar{\mathbb{N}}} = \text{unfold}_{\bar{\mathbb{N}}} \bar{\mathbb{N}} \hat{!} \text{out}_{\bar{\mathbb{N}}}$. Thanks to this coercion we can define streams of extended natural numbers.

One would like also to consider infinite trees labelled with elements in A . This could be defined using the functor $F(X) = A \otimes (X \otimes X)$. Unfortunately, the term defining this functor :

$$\lambda f. \lambda x. \text{let } \langle y, z \rangle = x \text{ in let } \langle u, v \rangle = z \text{ in } \langle y, \langle f u, f v \rangle \rangle$$

cannot be assigned a linear type because of the duplication of the variable f .

8. Related works

Infinite data structures in ICC Several works have studied properties related to ICC in the context of infinite data structures.

Burrell et al. [8] proposed Pola as a programming language characterizing FPTIME. The design idea of Pola comes from safe recursion on notation [7] and interestingly, Pola permits the programmer to write polynomial time functional programs working both on inductive and coinductive data types. This work is close to ours but there are two main differences. First, the use of safe recursion on notation and the use of linear types are quite different and produce two different programming methodologies. Second, we studied how to define algebras and coalgebras in the language while Pola takes inductive and coinductive types as primitive.

Leivant and Ramyaa [29] have studied a framework based on equational programs that is useful to reason about programs over inductive and coinductive types. They used such a framework to obtain an ICC characterization of primitive corecurrence (a weak form of productivity). Ramyaa and Leivant [36] also shows that a ramified version of corecurrence gives an ICC characterization of the class of functions over streams working in logarithmic space. Leivant and Ramyaa [30] further studied the correspondence between ramified recurrence and ramified corecurrence. In our work, in contrast we focus on the restrictions directly provided by Light Affine Logic. It can be an interesting future direction study whether one can express some form of ramified corecurrence in LAL, along the lines of what has been done for ramified recursion [34].

Using an approach based on quasi-interpretation, in [12, 13] we have studied space upper bounds properties and input-output properties of programs working on streams. Using a similar approach Férée et al. [10] showed that interpretations can be used on stream programs also to characterize type 2 polynomial time functions by providing a characterization of the class of the Basic Feasible Functionals of Cook and Urquhart [9].

Dal Lago et al. [3] have developed a technique inspired by quasi-interpretations to study the complexity of higher-order programs. In their framework infinite data are first class citizens in the form of higher order functions. However, they do not consider programs working on declarative infinite data structures as streams.

Expressivity of Light Logics Understanding and improving the expressivity of light logics, and more in general of ICC systems, is a well known problem. Unfortunately, we do not yet have a general method for comparing different systems and improvements.

Hofmann [25] provides a survey of the different approaches to ICC. In this survey he also discusses the expressivity and the limitations of the different light logics in the encoding of traditional algorithms. Murawski and Ong [34] and, more recently, Roversi and Vercelli [37] have studied the expressivity of light logics by comparing them with the one provided by ramified recursion. In particular, they provide different embeddings of (fragments of) ramified recursion in LLL and its extensions. Dal Lago et al. [28] have studied and compared the expressivity of different light logics obtained

by adding or removing several type constructions, like tensor product, polymorphism and type fixpoints. Our work follows in spirit the same approach focusing on the encoding of (co)algebras.

Gaboardi et al. [17] have studied the expressivity of the different light logics by designing embeddings from the light logics to Linear Logic by Level [4], another logic providing a characterization of polynomial time but based on more general principles. Interestingly, in Linear Logic by Level the \S modality commutes with all the other type constructions. It would be interesting to study what is the expressivity of this logic with respect to the encoding of algebras and coalgebras. Baillot et al. [6] have approached the problem of improving the expressivity of LAL by designed a programming language with recursion and pattern matching around it. We take inspiration by their work but instead of adding extra constructions we focus on the constructions that can be defined in LAL itself.

9. Conclusion and future works

We have studied the definability of algebras and coalgebras in the LALC along the lines of the encoding of algebras and coalgebras in the polymorphic lambda calculus. By extending the calculus with distributive rules for the modality \S we are able to program several natural examples over infinite data structures.

It is well-known that the encoding of algebras and coalgebras in System F is rather limited and it also does not behave well from the type theory point of view [18]. For this reason, several works have studied how to extend the polymorphic lambda calculus with different notions of algebras and coalgebras that behave better, e.g. [31]. We expect that a similar approach can be also followed for LALC: it would be interesting to understand how the different extensions studied in the literature can fit the LALC setting.

We have approached the study of algebras and coalgebras in a term language for Light Affine Logic (LALC). There are also other light logics for which we could ask the same question. Obviously, the an encoding similar to the one we studied here can be used for Elementary Linear Logic. It would instead be more interesting to understand whether there is an encoding for Soft Linear Logic that allows a large class of coinductive data structures to be defined.

References

- [1] A. Asperti. Light affine logic. In *LICS '98*, pages 300–308. IEEE, 1998.
- [2] A. Asperti and L. Roversi. Intuitionistic light affine logic. *ACM TOCL*, 3(1):137–175, 2002.
- [3] P. Baillot and U. Dal Lago. Higher-Order Interpretations and Program Complexity. In *CSL'12*, volume 16, pages 62–76. LIPIcs, 2012.
- [4] P. Baillot and D. Mazza. Linear logic by levels and bounded time complexity. *TCS*, 411(2):470–503, 2010.
- [5] P. Baillot and K. Terui. Light types for polynomial time computation in lambda-calculus. In *LICS '04*, pages 266–275. IEEE, 2004.
- [6] P. Baillot, M. Gaboardi, and V. Mogbil. A polytime functional language from light linear logic. In *ESOP'10*, volume 6012 of *LNCS*. Springer, 2010.
- [7] S. Bellantoni and S. Cook. A new recursion-theoretic characterization of the poly-time functions. *Comp. Complexity*, 2:97–110, 1992.
- [8] M. J. Burrell, R. Cockett, and B. F. Redmond. Pola: a language for PTIME programming. In *LCC'09*, 2009.
- [9] S. Cook and A. Urquhart. Functional interpretations of feasibly constructive arithmetic. *STOC '89*, pages 107–112. ACM, 1989.
- [10] H. Férée, E. Hainry, M. Hoyrup, and R. Péchoux. Interpretation of stream programs: characterizing type 2 polynomial time complexity. In *ISAAC'10*, pages 291–303. Springer, 2010.
- [11] P. Freyd. Structural polymorphism. *TCS*, 115(1):107–129, 1993.
- [12] M. Gaboardi and R. Péchoux. Upper bounds on stream I/O using semantic interpretations. In *CSL '09*, volume 5771 of *LNCS*, pages 271 – 286. Springer, 2009.
- [13] M. Gaboardi and R. Péchoux. Global and local space properties of stream programs. In *FOPARA'09*, volume 6324 of *LNCS*, pages 51 – 66. Springer, 2010.
- [14] M. Gaboardi and S. Ronchi Della Rocca. A soft type assignment system for λ -calculus. In *CSL '07*, volume 4646 of *LNCS*, pages 253–267. Springer, 2007.
- [15] M. Gaboardi, J.-Y. Marion, and S. Ronchi Della Rocca. Soft linear logic and polynomial complexity classes. In *LSFA 2007*, volume 205 of *ENTCS*, pages 67–87. Elsevier.
- [16] M. Gaboardi, J.-Y. Marion, and S. Ronchi Della Rocca. A logical account of PSPACE. In *POPL'08*. ACM, 2008.
- [17] M. Gaboardi, L. Roversi, and L. Vercelli. A by-level analysis of Multiplicative Exponential Linear Logic. In *MFCS'09*, volume 5734 of *LNCS*, pages 344 – 355. Springer, 2009.
- [18] H. Geuvers. Inductive and coinductive types with iteration and recursion. In *Types for Proofs and Programs*, pages 193–217. 1992.
- [19] J. Girard. *The Blind Spot: Lectures on Logic*. European Mathematical Society, 2011. ISBN 9783037190883.
- [20] J.-Y. Girard. Linear logic. *TCS*, 50:1–102, 1987.
- [21] J.-Y. Girard. Light linear logic. *Information and Computation*, 143(2): 175–204, 1998.
- [22] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge tracts in theoretical computer science*. Cambridge university press, 1989.
- [23] T. Hagino. A typed lambda calculus with categorical type constructors. In *Category Theory and Computer Science*, volume 283, pages 140–57. LNCS, 1987.
- [24] P. Hájek. Arithmetical hierarchy and complexity of computation. *TCS*, 8:227–237, 1979.
- [25] M. Hofmann. Programming languages capturing complexity classes. *ACM SIGACT News*, 31(1):31–42, 2000.
- [26] B. Jacobs and J. Rutten. A tutorial on (co) algebras and (co) induction. *EATCS*, 62:222–259, 1997.
- [27] Y. Lafont. Soft linear logic and polynomial time. *TCS*, 318(1-2):163–180, 2004.
- [28] U. D. Lago and P. Baillot. On light logics, uniform encodings and polynomial time. *MSCS'06*, 16(4):713–733, 2006.
- [29] D. Leivant and R. Ramyaa. Implicit complexity for coinductive data: a characterization of corecurrence. In *DICE'12*, volume 75 of *EPTCS*, pages 1–14, 2012.
- [30] D. Leivant and R. Ramyaa. The computational contents of ramified corecurrence. In *FOSSACS*, 2015. To appear.
- [31] R. Matthes. Monotone (co)inductive types and positive fixed-point types. *ITA*, 33(4/5):309–328, 1999.
- [32] F. Maurel. Nondeterministic light logics and NP-time. In *TLCA '03*, volume 2701 of *LNCS*, pages 241–255. Springer, 2003.
- [33] D. Mazza. Non-uniform polytime computation in the infinitary affine lambda-calculus. In *ICALP'14*, pages 305–317, 2014.
- [34] A. S. Murawski and C. L. Ong. On an interpretation of safe recursion in light affine logic. *TCS*, 318(1-2):197–223, 2004.
- [35] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
- [36] R. Ramyaa and D. Leivant. Ramified corecurrence and logspace. *ENTCS*, 276(0):247 – 261, 2011. MFPS'11.
- [37] L. Roversi and L. Vercelli. Safe recursion on notation into a light logic by levels. In *DICE'10*, pages 63–77, 2010.
- [38] U. Schöpp. Stratified bounded affine logic for logarithmic space. In *LICS '07*, pages 411–420. Washington, DC, USA, 2007. IEEE.
- [39] K. Terui. Light affine lambda calculus and polytime strong normalization. In *LICS '01*, pages 209–220. IEEE, 2001.
- [40] P. Wadler. Recursive types for free! Technical report, University of Glasgow, 1990.
- [41] G. C. Wraith. A note on categorical datatypes. In *CTCS*, volume 389 of *LNCS*, pages 118–127. Springer, 1993.

4 Characterizing polytime complexity of stream programs using interpretations

Characterizing polynomial time complexity of stream programs using interpretations

Hugo Férée, Emmanuel Hainry, Mathieu Hoyrup, Romain Péchoux

► **To cite this version:**

Hugo Férée, Emmanuel Hainry, Mathieu Hoyrup, Romain Péchoux. Characterizing polynomial time complexity of stream programs using interpretations. Theoretical Computer Science, Elsevier, 2015, 585, pp.41-54. 10.1016/j.tcs.2015.03.008 . hal-01112160

HAL Id: hal-01112160

<https://hal.inria.fr/hal-01112160>

Submitted on 2 Feb 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Characterizing polynomial time complexity of stream programs using interpretations

Hugo Férée^{a,c}, Emmanuel Hainry^{a,c}, Mathieu Hoyrup^{b,c}, Romain Péchoux^{a,c}

^a *Université de Lorraine, Nancy, France*

^b *Inria Nancy - Grand Est, Villers-lès-Nancy, France*

^c *Project-team CARTE, LORIA, UMR7503*

Abstract

This paper provides a criterion based on interpretation methods on term rewrite systems in order to characterize the polynomial time complexity of second order functionals. For that purpose it introduces a first order functional stream language that allows the programmer to implement second order functionals. This characterization is extended through the use of exp-poly interpretations as an attempt to capture the class of Basic Feasible Functionals (BFF). Moreover, these results are adapted to provide a new characterization of polynomial time complexity in computable analysis. These characterizations give a new insight on the relations between the complexity of functional stream programs and the classes of functions computed by Oracle Turing Machine, where oracles are treated as inputs.

Keywords:

Stream Programs, Type-2 Functionals, Interpretations, Polynomial Time, Basic Feasible Functionals, Computable Analysis, Rewriting

1. Introduction

Lazy functional languages like Haskell allow the programmer to deal with co-inductive datatypes in such a way that co-inductive objects can be evaluated by finitary means. Consequently, computations over streams, that is infinite lists, can be performed in such languages.

A natural question arising is the complexity of the programs computing on streams. Intuitively, the complexity of a stream program is the number of reduction steps needed to output the n first elements of a stream, for any n . However the main issue is to relate the complexity bound to the input structure. Since a stream can be easily identified with a function, a good way for solving such an issue is to consider computational and complexity models dealing with functions as inputs.

In this perspective, we want to take advantage of the complexity results obtained on type-2 functions (functions over functions), and in particular on BFF [1, 2], to understand stream program complexity. For that purpose, we

set Unary Oracle Turing Machine (UOTM), machines computing functions with oracles taking unary inputs, as our main computational model. This model is well-suited in our framework since it manipulates functions as objects with a well-defined notion of complexity. UOTM are a refinement of Oracle Turing Machines (OTM) on binary words which correspond exactly to the BFF algebra in [3] under polynomial restrictions. UOTM are better suited than OTM to study stream complexity with a realistic complexity measure, since in the UOTM model accessing the n^{th} element costs n transitions whereas it costs $\log(n)$ in the OTM model.

The Implicit Computational Complexity (ICC) community has proposed characterizations of OTM complexity classes using function algebra [3, 4] and type systems [5, 6] or recently as a logic [7].

These latter characterizations are inspired by former characterizations of type-1 polynomial time complexity based on ramification [8, 9]. This line of research has led to new developments of other ICC tools and in particular to the use of (polynomial) interpretations in order to characterize the classes of functions computable in polynomial time or space [10, 11].

Polynomial interpretations [12, 13] are a well-known tool used to show the termination of first order term rewrite systems. This tool has been adapted into variants, like quasi-interpretations and sup-interpretation [14], that allow the programmer to analyze program complexity. In general, interpretations are restricted to inductive data types and [15] was a first attempt to adapt such a tool to co-inductive data types including stream programs. In this paper, we introduce a second order variation of this interpretation methodology in order to constrain the complexity of stream program computation and we obtain a characterization of UOTM polynomial time computable functions. Using this characterization, we can analyze functions of this class in an easier way based on the premise that it is practically easier to write a first order functional program on streams than the corresponding Unary Oracle Turing Machine. The drawback is that the tool suffers from the same problems as polynomial interpretation: the difficulty to automatically synthesize the interpretation of a given program (see [16]). As a proof of versatility of this tool, we provide a partial characterization of the BFF class (the full characterization remaining open), just by changing the interpretation codomain: for that purpose, we use restricted exponentials instead of polynomials in the interpretation of a stream argument.

A direct and important application is that second order polynomial interpretations on stream programs can be adapted to characterize the complexity of functions computing over reals defined in Computable Analysis [17]. This approach is a first attempt to study the complexity of such functions through static analysis methods.

This paper is an extended version of [18] with complete proofs, additional examples and corrections.

Outline of the paper.

In Section 2, we introduce (Unary) Oracle Turing Machines and their complexity. In Section 3, we introduce the studied first order stream language. In

Section 4, we define the interpretation tools extended to second order and we provide a criterion on stream programs. We show our main characterization relying on the criterion in Section 5. Section 6 develops a new application, which was only mentioned in [18], to functions computing over reals.

2. Polynomial time Oracle Turing Machines

In this section, we will define a machine model and a notion of complexity relevant for stream computations. This model (UOTM) is adapted from the Oracle Turing Machine model used by Kapron and Cook in their characterization of Basic Feasible functionals (BFF) [3]. In the following, $|x|$ will denote the size of the binary encoding of $x \in \mathbb{N}$, namely $\lceil \log_2(x) \rceil$.

Definition 1 (Oracle Turing Machine). An Oracle Turing Machine (denoted by OTM) \mathcal{M} with k oracles and l input tapes is a Turing machine with, for each oracle, a state, one query tape and one answer tape.

Whenever \mathcal{M} is used with input oracles $F_1, \dots, F_k : \mathbb{N} \rightarrow \mathbb{N}$ and arrives on the oracle state $i \in \{1, \dots, k\}$ and if the corresponding query tape contains the binary encoding of a number x , then the binary encoding of $F_i(x)$ is written on the corresponding answer tape. It behaves like a standard Turing machine on the other states.

We now introduce the unary variant of this model, which is more related to stream computations as accessing the n -th element takes at least n steps (whereas it takes $\log(n)$ steps in OTM. See example 2 for details).

Definition 2 (Unary Oracle Turing Machine). A Unary Oracle Turing Machine (denoted UOTM) is an OTM where numbers are written using unary notation on the query tape, *i.e.* on the oracle state i , if w is the content of the corresponding query tape, then $F_i(|w|)$ is written on the corresponding answer tape.

Definition 3 (Running time). In both cases (OTM and UOTM), we define the cost of a transition as the size of the answer of the oracle, in the case of a query, and 1 otherwise.

In order to introduce a notion of complexity, we have to define the size of the inputs of our machines.

Definition 4 (Size of a function). The size $|F| : \mathbb{N} \rightarrow \mathbb{N}$ of a function $F : \mathbb{N} \rightarrow \mathbb{N}$ is defined by:

$$|F|(n) = \max_{k \leq n} |F(k)|$$

Remark 1. This definition is different from the one used in [3] (denoted here by $\|\cdot\|$). Indeed, the size of a function was defined by $\|F\|(n) = \max_{|k| \leq n} |F(k)|$, in other words, $\|f\|(n) = |f|(2^n - 1)$. The reason for this variation is that in an UOTM, the oracle is closer to an infinite sequence (or stream as we will see in the

following) than to a function, since it can access easily its first n elements but not its $(2^n)^{th}$ element which is the case in an OTM. In particular, this makes the size function computable in polynomial time (with respect to the following definitions).

The size of an oracle input is then a type-1 function whereas it is an integer for standard input. Then, the notion of polynomial running time needs to be adapted.

Definition 5 (Second order polynomial). A second order polynomial is a polynomial generated by the following grammar:

$$P := c \mid X \mid P + P \mid P \times P \mid Y\langle P \rangle$$

where X represents a zero order variable (ranging over \mathbb{N}), Y a first order variable (ranging over $\mathbb{N} \rightarrow \mathbb{N}$), c a constant in \mathbb{N} and $\langle - \rangle$ is a notation for the functional application.

Example 1. $P(Y_1, Y_2, X_1, X_2) = (Y_1\langle Y_1\langle X_1 \rangle \times Y_2\langle X_2 \rangle \rangle)^2$ is a second order polynomial.

The following lemma shows that the composition of polynomials is well behaved.

Lemma 1. *If the first order variables of a second order polynomial are replaced with first order polynomials, then the resulting function is a first order polynomial.*

PROOF. This can be proved by induction on the definition of a second order polynomial: this is true if P is constant or equal to a zero order variable and also if P is a sum, a multiplication or a composition with type-1 variable, since polynomials are stable under these operators.

In the following, $P(Y_1, \dots, Y_k, X_1, \dots, X_l)$ will denote a second order polynomial where each Y_i represents a first order variable, and each X_i a zero order variable.

This definition of running time is directly adapted from the definition of running time for OTMs.

Definition 6 (Polynomial running time). An UOTM \mathcal{M} operates in time $T : (\mathbb{N} \rightarrow \mathbb{N})^k \rightarrow \mathbb{N}^l \rightarrow \mathbb{N}$ if for all inputs $x_1, \dots, x_l : \mathbb{N}$ and $F_1, \dots, F_k : \mathbb{N} \rightarrow \mathbb{N}$, the sum of the transition costs before \mathcal{M} halts on these inputs is less than $T(|F_1|, \dots, |F_k|, |x_1|, \dots, |x_l|)$.

A function $G : (\mathbb{N} \rightarrow \mathbb{N})^k \rightarrow \mathbb{N}^l \rightarrow \mathbb{N}$ is UOTM computable in polynomial time if there exists a second order polynomial P such that G is computed by an UOTM in time P .

The class BFF is defined in a similar manner substituting $||\cdot||$ to $|\cdot|$.

Lemma 2. *The set of polynomial time UOTM computable functions is strictly included in the set of polynomial time OTM computable functions (proved to be equal to the BFF algebra [3]):*

PROOF. In order to transform a UOTM into an OTM computing the same functional, we have to convert the content of the query tape from the word w (representing $|w|$ in unary) into the binary encoding of $|w|$ before each oracle call. This can be done in polynomial time in $|w|$ and we call Q this polynomial. In both cases, the cost of the transition is $|F(|w|)|$, so the running time of both machines is the same, except for the conversion time. If the computation time of the UOTM was bounded by a second order polynomial P , the size of the content of the query is at most $P(|F|)$ at each query, so the additional conversion time is at most $Q(P(|F|))$ for each query and there are at most $P(|F|)$ queries. Thus, the conversion time is also a second order polynomial in the size of the inputs, and the computation time of the OTM is also bounded by a second order polynomial in $|F|$. Finally, since $\|F\|$ bounds $|F|$, the computation time is also bounded by the same polynomial in $\|F\|$, so F is in BFF.

Example 2. The function $G : (\mathbb{N} \rightarrow \mathbb{N}) \times \mathbb{N} \rightarrow \mathbb{N}$ defined by $G(F, x) = F(|x|) = F(\lceil \log_2(x) \rceil)$ is UOTM computable in polynomial time. Indeed, it is computed by the function which copies the input word representing x on the query tape before entering the query state before returning the content of the answer tape as output. Its running time is bounded by $2 \times (|x| + |F|(|x|))$.

However $H(F, x) = F(x)$ is not UOTM computable in polynomial time, because it would require to write x in unary on the query tape, which costs $2^{|x|}$. Nonetheless, H is in BFF because an OTM only has to write x in binary, and the oracle call costs $|F(x)| \leq \|F\|(|x|)$.

3. First order stream language

3.1. Syntax

In this section, we define a simple Haskell-like lazy first order functional language with streams.

\mathcal{F} will denote the set of function symbols, \mathcal{C} the set of constructor symbols and \mathcal{X} the set of variable names. A program is a list of definitions \mathcal{D} given by the grammar in Figure 1:

$$\begin{array}{ll}
 p ::= x \mid c \ p_1 \ \dots \ p_n \mid p : y & \text{(Patterns)} \\
 e ::= x \mid t \ e_1 \ \dots \ e_n & \text{(Expressions)} \\
 d ::= f \ p_1 \ \dots \ p_n = e & \text{(Definitions)}
 \end{array}$$

Figure 1: Program grammar

where $x, y \in \mathcal{X}$, $t \in \mathcal{C} \cup \mathcal{F}$, $c \in \mathcal{C} \setminus \{:\}$ and $f \in \mathcal{F}$ and c, t and f are symbols of arity n .

Throughout the paper, we call closed expression any expression without variables.

The stream constructor $:$ $\in \mathcal{C}$ is a special infix constructor of arity 2. In a stream expression $h : t$, h and t are respectively called the head and the tail of the stream.

We might use other infix or postfix constructors or function symbols in the following for ease of readability.

In a definition $f p_1 \dots p_n = e$, all the variables of e appear in the patterns p_i . Moreover patterns are non overlapping and each variable appears at most once in the left-hand side. This entails that programs are confluent. In a program, we suppose that all pattern matchings are exhaustive. Finally, we only allow patterns of depth 1 for the stream constructor (*i.e.* only variables appear in the tail of a stream pattern).

Remark 2. This is not restrictive since a program with higher pattern matching depth can be easily transformed into a program of this form using extra function symbols and definitions. We will prove in Proposition 1 that this modification does not alter our results on polynomial interpretations.

3.2. Type system

Programs contain inductive types that will be denoted by Tau . For example, unary integers are defined by:

```
data Nat = 0 | Nat +1
```

(with $0, +1 \in \mathcal{C}$), and binary words by:

```
data Bin = Nil | 0 Bin | 1 Bin.
```

Consequently, each constructor symbol comes with a typed signature and we will use the notation $c :: T$ to denote that the constructor symbol c has type T . For example, we have $0 :: \text{Nat}$ and $+1 :: \text{Nat} \rightarrow \text{Nat}$. Note that in the following, given some constants $n, k, \dots \in \mathbb{N}$, the terms n, k, \dots denote their encoding as unary integers in Nat .

Programs contain co-inductive types defined by `data [Tau] = Tau : [Tau]` for each inductive type Tau . This is a distinction with Haskell, where streams are defined to be both finite and infinite lists, but not a restriction since finite lists may be defined in this language and since we are only interested in showing properties of total functions (*i.e.* an infinite stream represents a total function).

In the following, we will write T^k for $T \rightarrow T \rightarrow \dots \rightarrow T$ (with k occurrences of T).

Each function symbol f comes with a typed signature that we restrict to be either $f :: [\text{Tau}]^k \rightarrow \text{Tau}^l \rightarrow \text{Tau}$ or $f :: [\text{Tau}]^k \rightarrow \text{Tau}^l \rightarrow [\text{Tau}]$, with $k, l \geq 0$.

Throughout the paper, we will only consider well-typed programs where the left-hand side and the right-hand side of a definition can be given the same type using the simple rules of Figure 2 with $A, A_i \in \{\text{Tau}, [\text{Tau}]\}$. Γ is a typing basis for every variable, constructor and function symbol (we assume that each variable is used in at most one definition, for the sake of simplicity).

$$\frac{\Gamma(x) = A}{\Gamma \vdash x :: A} \quad x \in \mathcal{X} \cup \mathcal{C} \cup \mathcal{F}$$

$$\frac{\forall (f \ p_1 \dots p_n = e) \in \mathcal{D}, \quad \Gamma \vdash e :: A \quad \forall i, \Gamma \vdash p_i :: A_i}{\Gamma \vdash f :: A_1 \rightarrow \dots \rightarrow A_n \rightarrow A} \quad f \in \mathcal{F}$$

$$\frac{\Gamma \vdash t :: A_1 \rightarrow \dots \rightarrow A_n \rightarrow A \quad \forall i \in \{1, \dots, n\}, \ e_i :: A_i}{\Gamma \vdash t \ e_1 \dots e_n :: A} \quad t \in \mathcal{C} \cup \mathcal{F}$$

Figure 2: Typing rules

3.3. Semantics

Lazy values and strict values are defined in Figure 3:

$$\begin{aligned} \text{lv} &::= e_1 : e_2 && \text{(Lazy value)} \\ \text{v} &::= c \ v_1 \dots v_n && \text{(Strict value)} \end{aligned}$$

Figure 3: Values and lazy values

where e_1, e_2 are closed expressions and c belongs to $\mathcal{C} \setminus \{:\}$. Lazy values are stream expressions with the constructor symbol $:$ at the top level whereas strict values are expressions of inductive type where only constructor symbols occur and are used to deal with fully evaluated elements.

Moreover, let \mathfrak{S} represent the set of substitutions σ that map variables to expressions. As usual the result of applying the substitution σ to an expression e is denoted $\sigma(e)$.

The evaluation rules are defined in Figure 4:

$$\frac{(\mathbf{f} \ p_1 \ \dots \ p_n = \mathbf{e}) \in \mathcal{D} \quad \sigma \in \mathfrak{S} \quad \forall i \in \{1, \dots, n\}, \sigma(p_i) = \mathbf{e}_i}{\mathbf{f} \ \mathbf{e}_1 \ \dots \ \mathbf{e}_n \rightarrow \sigma(\mathbf{e})} \quad (d)$$

$$\frac{\mathbf{e}_i \rightarrow \mathbf{e}'_i \quad \mathbf{t} \in \mathcal{F} \cup \mathcal{C} \setminus \{:\}}{\mathbf{t} \ \mathbf{e}_1 \ \dots \ \mathbf{e}_i \ \dots \ \mathbf{e}_n \rightarrow \mathbf{t} \ \mathbf{e}_1 \ \dots \ \mathbf{e}'_i \ \dots \ \mathbf{e}_n} \quad (t)$$

$$\frac{\mathbf{e} \rightarrow \mathbf{e}'}{\mathbf{e} : \mathbf{e}_0 \rightarrow \mathbf{e}' : \mathbf{e}_0} \quad (:)$$

Figure 4: Evaluation rules

We will write $\mathbf{e} \rightarrow^n \mathbf{e}'$ if there exist expressions $\mathbf{e}_1, \dots, \mathbf{e}_{n-1}$ such that $\mathbf{e} \rightarrow \mathbf{e}_1 \cdots \rightarrow \mathbf{e}_{n-1} \rightarrow \mathbf{e}'$. Let \rightarrow^* denote the transitive and reflexive closure of \rightarrow . We write $\mathbf{e} \rightarrow_! \mathbf{e}'$ if \mathbf{e} is normalizing to the expression \mathbf{e}' , *i.e.* $\mathbf{e} \rightarrow^* \mathbf{e}'$ and there is no \mathbf{e}'' such that $\mathbf{e}' \rightarrow \mathbf{e}''$. We can show easily wrt the evaluation rules (and because definitions are exhaustive) that given a closed expression \mathbf{e} , if $\mathbf{e} \rightarrow_! \mathbf{e}'$ and $\mathbf{e} :: \text{Tau}$ then \mathbf{e}' is a strict value, whereas if $\mathbf{e} \rightarrow_! \mathbf{e}'$ and $\mathbf{e} :: [\text{Tau}]$ then \mathbf{e}' is a lazy value. Indeed the (t) rule of Figure 4 allows the reduction of an expression under a function or constructor symbol whereas the (:) rule only allows reduction of a stream head (this is why stream patterns of depth greater than 1 are not allowed in a definition).

These reduction rules are not deterministic but a call-by-need strategy could be defined to mimic Haskell's semantic.

The following function symbols are typical stream operators and will be used further.

Example 3. $\mathbf{s} \ !! \ \mathbf{n}$ computes the $(n + 1)^{th}$ element of the stream \mathbf{s} :

```
!! :: [Tau] -> Nat -> Tau
(h:t) !! (n+1) = t !! n
(h:t) !! 0 = h
```

Example 4. $\mathbf{tln} \ \mathbf{s} \ \mathbf{n}$ drops the first $n + 1$ elements of the stream \mathbf{s} :

```
tln :: [Tau] -> Nat -> [Tau]
tln (h:t) (n+1) = tln t n
tln (h:t) 0 = t
```

4. Second order polynomial interpretations

In the following, we will call a **positive functional** any function of type $(\mathbb{N} \rightarrow \mathbb{N})^k \rightarrow \mathbb{N}^l \rightarrow T$ with $k, l \in \mathbb{N}$ and $T \in \{\mathbb{N}, \mathbb{N} \rightarrow \mathbb{N}\}$.

Given a positive functional $F : ((\mathbb{N} \rightarrow \mathbb{N})^k \times \mathbb{N}^l) \rightarrow T$, the arity of F is $k + l$.

Let $>$ denote the usual ordering on \mathbb{N} and its standard extension to $\mathbb{N} \rightarrow \mathbb{N}$, *i.e.* given $F, G : \mathbb{N} \rightarrow \mathbb{N}$, $F > G$ if $\forall n \in \mathbb{N} \setminus \{0\}, F(n) > G(n)$ (the comparison on 0 is not necessary since we will only use strictly positive inputs).

We extend this ordering to the set of positive functionals of arity $k + l$ by: $F > G$ if

$$\begin{aligned} \forall y_1, \dots, y_k \in \mathbb{N} \rightarrow^{\uparrow} \mathbb{N}, \forall x_1 \dots x_l \in \mathbb{N} \setminus \{0\}, \\ F(y_1, \dots, y_k, x_1, \dots, x_l) > G(y_1, \dots, y_k, x_1, \dots, x_l) \end{aligned}$$

where $\mathbb{N} \rightarrow^{\uparrow} \mathbb{N}$ is the set of increasing functions on positive integers.

Definition 7 (Monotonic positive functionals). A positive functional is monotonic if it is strictly increasing with respect to each of its arguments.

Definition 8 (Types interpretation). The type signatures of a program are interpreted as types this way:

- an inductive type \mathbf{Tau} is interpreted as \mathbb{N}
- a stream type $[\mathbf{Tau}]$ is interpreted as $\mathbb{N} \rightarrow \mathbb{N}$
- an arrow type $\mathbf{A} \rightarrow \mathbf{B}$ is interpreted as the type $T_{\mathbf{A}} \rightarrow T_{\mathbf{B}}$ if \mathbf{A} and \mathbf{B} are respectively interpreted as $T_{\mathbf{A}}$ and $T_{\mathbf{B}}$.

Definition 9 (Assignment). An assignment is a mapping of each function symbol $\mathbf{f} :: \mathbf{A}$ to a monotonic positive functional whose type is the interpretation of \mathbf{A} .

An assignment can be canonically extended to any expression in the program:

Definition 10 (Expression assignment). The assignment is defined on constructors and expressions this way:

- $\langle \mathbf{c} \rangle (X_1, \dots, X_n) = \sum_{i=1}^n X_i + 1$, if $\mathbf{c} \in \mathcal{C} \setminus \{:\}$ is of arity n .
- $\begin{cases} \langle \mathbf{:} \rangle (X, Y)(0) = X \\ \langle \mathbf{:} \rangle (X, Y)(Z + 1) = 1 + X + Y \langle Z \rangle \end{cases}$
- $\langle \mathbf{x} \rangle = X$, if \mathbf{x} is a variable of type \mathbf{Tau} , *i.e.* we associate a unique zero order variable X in \mathbb{N} to each $\mathbf{x} \in \mathcal{X}$ of type \mathbf{Tau} .
- $\langle \mathbf{y} \rangle (Z) = Y \langle Z \rangle$, if \mathbf{y} is a variable of type $[\mathbf{Tau}]$, *i.e.* we associate a unique first order variable $Y : \mathbb{N} \rightarrow \mathbb{N}$ to each $\mathbf{y} \in \mathcal{X}$ of type $[\mathbf{Tau}]$.
- $\langle \mathbf{t} \mathbf{e}_1 \dots \mathbf{e}_n \rangle = \langle \mathbf{t} \rangle (\langle \mathbf{e}_1 \rangle, \dots, \langle \mathbf{e}_n \rangle)$, if $\mathbf{t} \in \mathcal{C} \cup \mathcal{F}$.

Remark 3. For every expression e , $\llbracket e \rrbracket$ is a monotonic positive functional, since it is true for function symbols, for constructors, and the composition of such functionals is still monotonic positive.

Example 5. The stream constructor $:$ has type $\mathbf{Tau} \rightarrow [\mathbf{Tau}] \rightarrow [\mathbf{Tau}]$. Consequently, its assignment $\llbracket : \rrbracket$ has type $(\mathbb{N} \times (\mathbb{N} \rightarrow \mathbb{N})) \rightarrow (\mathbb{N} \rightarrow \mathbb{N})$. Let us consider the expression $p : (q : r)$, with $p, q, r \in \mathcal{X}$, we obtain that:

$$\begin{aligned} \llbracket p : (q : r) \rrbracket &= \llbracket : \rrbracket(\llbracket p \rrbracket, \llbracket q : r \rrbracket) = \llbracket : \rrbracket(\llbracket p \rrbracket, \llbracket : \rrbracket(\llbracket q \rrbracket, \llbracket r \rrbracket)) = \llbracket : \rrbracket(P, \llbracket : \rrbracket(Q, R)) \\ &= F(R, Q, P) \end{aligned}$$

where $F \in ((\mathbb{N} \rightarrow \mathbb{N}) \times \mathbb{N}^2) \rightarrow (\mathbb{N} \rightarrow \mathbb{N})$ is the positive functional such that:

- $F(R, Q, P)(Z + 2) = 1 + P + \llbracket : \rrbracket(Q, R)(Z + 1) = 2 + P + Q + R(Z)$
- $F(R, Q, P)(1) = 1 + P + \llbracket : \rrbracket(Q, R)(0) = 1 + P + Q$
- $F(R, Q, P)(0) = P$

Lemma 3. *The assignment of an expression e defines a positive functional in the assignment of its free variables (and an additional type-0 variable if $e :: [\mathbf{Tau}]$).*

PROOF. By structural induction on expressions. This is the case for variables and for the stream constructor, the inductive constructors are additive, and the assignments of function symbols are positive (the composition of a positive functional with positive functional being a positive functional).

Definition 11 (Polynomial interpretation). An assignment $\llbracket - \rrbracket$ of the function symbols of a program defines an interpretation if for each definition $f \ p_1 \dots p_n = e \in \mathcal{D}$,

$$\llbracket f \ p_1 \dots p_n \rrbracket > \llbracket e \rrbracket.$$

Furthermore, $\llbracket f \rrbracket$ is polynomial if it is bounded by a second order polynomial. In this case, the program is said to be polynomial.

The following programs will be used further and have polynomial interpretations:

Example 6. The sum over unary integers:

```
plus :: Nat -> Nat -> Nat
plus 0 b = b
plus (a+1) b = (plus a b)+1
```

plus admits the following (polynomial) interpretation:

$$\llbracket \mathbf{plus} \rrbracket(X_1, X_2) = 2 \times X_1 + X_2$$

Indeed, we check that the following inequalities are satisfied:

$$\begin{aligned} \llbracket \mathbf{plus} \ 0 \ b \rrbracket &= 2 + B > B = \llbracket \mathbf{b} \rrbracket \\ \llbracket \mathbf{plus} \ (a+1) \ b \rrbracket &= 2A + 2 + B > 2A + B + 1 = \llbracket (\mathbf{plus} \ a \ b) + 1 \rrbracket \end{aligned}$$

Example 7. The product of unary integers:

```

mult :: Nat -> Nat -> Nat
mult 0 b = 0
mult (a+1) b = plus b (mult a b)

```

The following inequalities show that $\langle \text{mult} \rangle(X_1, X_2) = 3 \times X_1 \times X_2$ is an interpretation for `mult`:

$$\begin{aligned}
\langle \text{mult } 0 \text{ b} \rangle &= 3 \times \langle 0 \rangle \times \langle \text{b} \rangle = 3 \times B > 1 = \langle 0 \rangle \\
\langle \text{mult } (a+1) \text{ b} \rangle &= 3 \times A \times B + 3 \times B \\
&> 2 \times B + 3 \times A \times B = \langle \text{plus } b \text{ (mult } a \text{ b)} \rangle
\end{aligned}$$

Note that these interpretations are first order polynomials because `plus` and `mult` only have inductive arguments.

Example 8. The function symbol `!!` defined in Example 3 admits an interpretation of type $(\mathbb{N} \rightarrow \mathbb{N}) \times \mathbb{N} \rightarrow \mathbb{N}$ defined by:

$$\langle \text{!!} \rangle(Y, N) = Y \langle N \rangle.$$

Indeed, we check that:

$$\begin{aligned}
\langle \langle \text{h:t} \rangle \text{ !! } \langle \text{n+1} \rangle \rangle &= \langle \text{h:t} \rangle(\langle \text{n} \rangle + 1) = 1 + \langle \text{h} \rangle + \langle \text{t} \rangle(\langle \text{n} \rangle) > \langle \text{t} \rangle(\langle \text{n} \rangle) = \langle \text{t} \text{ !! } \text{n} \rangle \\
\langle \langle \text{h:t} \rangle \text{ !! } 0 \rangle &= \langle \text{h:t} \rangle(\langle 0 \rangle) = \langle \text{h:t} \rangle(1) = 1 + \langle \text{h} \rangle + \langle \text{t} \rangle(0) > \langle \text{h} \rangle
\end{aligned}$$

Example 9. The function symbol `tln` defined in Example 4 admits an interpretation of type $(\mathbb{N} \rightarrow \mathbb{N}) \times \mathbb{N} \rightarrow (\mathbb{N} \rightarrow \mathbb{N})$ defined by:

$$\langle \text{tln} \rangle(Y, N)(Z) = Y \langle N + Z + 1 \rangle.$$

Indeed:

$$\langle \text{tln } \langle \text{h:t} \rangle \text{ } \langle \text{n+1} \rangle \rangle(Z) = 1 + H + T(N + Z) > T(N + Z) = \langle \text{tln } \text{t } \text{n} \rangle(Z)$$

Theorem 1. *The synthesis problem, i.e. does a program admit a polynomial interpretation, is undecidable.*

PROOF. It has already been proved in [19] that the synthesis problem is undecidable for usual integer polynomials and such term rewriting systems without streams. Since our programs include them, and that their interpretations are necessarily first order integer polynomials, the synthesis problem for our stream programs is also undecidable.

Now that we have polynomial interpretations for `!!` and `tln`, we can prove that restricting to stream patterns of depth 1 was not a loss of generality.

Proposition 1. *If a program with arbitrary depth on stream patterns has a polynomial interpretation, then there exists an equivalent depth-1 program with a polynomial interpretation.*

PROOF. If \mathbf{f} has one stream argument using depth $k + 1$ pattern matching in a definition of the shape $\mathbf{f}(\mathbf{p}_1 : (\mathbf{p}_2 : \dots (\mathbf{p}_{k+1} : \mathbf{t}) \dots)) = \mathbf{e}$, we transform \mathbf{f} in a new equivalent program using $!!$ and \mathbf{tln} defined in Examples 3 and 4, and an auxiliary function symbol \mathbf{f}_1

$$\begin{aligned} \mathbf{f} \ \mathbf{s} &= \mathbf{f}_1 \ (\mathbf{tln} \ \mathbf{s} \ \mathbf{k}) \ (\mathbf{s} \ !! \ 0) \ \dots \ (\mathbf{s} \ !! \ \mathbf{k}) \\ \mathbf{f}_1 \ \mathbf{t} \ \mathbf{p}_1 \ \dots \ \mathbf{p}_{k+1} &= \mathbf{e} \end{aligned}$$

This new definition of \mathbf{f} is equivalent to the initial one, and if it admitted a polynomial interpretation P , then the equivalent function symbol \mathbf{f}_1 can be interpreted by:

$$\langle \mathbf{f}_1 \rangle (T, X_1, \dots, X_{k+1})(Z) = P\left(\sum_{1 \leq i \leq k+1} X_i + T(Z) + k + 1\right).$$

Then, we redefine $\langle \mathbf{f} \rangle$ by:

$$\langle \mathbf{f} \rangle (S)(Z) = 1 + \langle \mathbf{f}_1 \rangle (S, S(1), S(2), \dots, S(k+1))(Z)$$

which is greater than:

$$\langle \mathbf{f}_1 \rangle (\langle \mathbf{s} \rangle, \langle \mathbf{s} \ !! \ 0 \rangle, \dots, \langle \mathbf{s} \ !! \ (\mathbf{k}) \rangle)(Z)$$

so it is indeed a polynomial interpretation for this modified \mathbf{f} .

Lemma 4. *If \mathbf{e} is an expression of a program with an interpretation $\langle - \rangle$ and $\mathbf{e} \rightarrow \mathbf{e}'$, then $\langle \mathbf{e} \rangle > \langle \mathbf{e}' \rangle$.*

PROOF. If $\mathbf{e} \rightarrow \mathbf{e}'$ using:

- the (d) rule, then there are a substitution σ and a definition $\mathbf{f} \ \mathbf{p}_1 \dots \mathbf{p}_n = \mathbf{d}$ such that $\mathbf{e} = \sigma(\mathbf{f} \ \mathbf{p}_1 \dots \mathbf{p}_n)$ and $\mathbf{e}' = \sigma(\mathbf{d})$. We obtain $\langle \mathbf{e} \rangle > \langle \mathbf{e}' \rangle$ by definition of an interpretation and since $\langle \cdot \rangle > \langle \cdot \rangle$ is stable by substitution.
- the (t) -rule, then $\langle \mathbf{t} \ \mathbf{e}_1 \dots \mathbf{e}_i \dots \mathbf{e}_n \rangle > \langle \mathbf{t} \ \mathbf{e}_1 \dots \mathbf{e}'_i \dots \mathbf{e}_n \rangle$ is obtained by definition of $>$ and since $\langle \mathbf{t} \rangle$ is monotonic, according to Remark 3.
- the $(:)$ -rule, then $\mathbf{e} = \mathbf{d} : \mathbf{d}_0$ and $\mathbf{e}' = \mathbf{d}' : \mathbf{d}_0$, for some \mathbf{d}, \mathbf{d}' such that $\mathbf{d} \rightarrow \mathbf{d}'$. By induction hypothesis, $\langle \mathbf{d} \rangle > \langle \mathbf{d}' \rangle$, so:

$$\begin{aligned} \forall Z \geq 1, \langle \mathbf{d} : \mathbf{d}_0 \rangle (Z) &= 1 + \langle \mathbf{d} \rangle + \langle \mathbf{d}_0 \rangle (Z - 1) \\ &> 1 + \langle \mathbf{d}' \rangle + \langle \mathbf{d}_0 \rangle (Z - 1) = \langle \mathbf{d}' : \mathbf{d}_0 \rangle (Z) \end{aligned}$$

Notice that it also works for the base case $Z = 0$ by definition of $\langle \cdot \rangle$.

Proposition 2. *Given a closed expression $\mathbf{e} :: \mathbf{Tau}$ of a program with an interpretation $\langle - \rangle$, if $\mathbf{e} \rightarrow^n \mathbf{e}'$ then $n \leq \langle \mathbf{e} \rangle$. In other words, every reduction chain starting from an expression \mathbf{e} of a program with interpretation has its length bounded by $\langle \mathbf{e} \rangle$.*

Corollary 1. *Given a closed expression $e :: [\text{Tau}]$ of a program with an interpretation $(-)$, if $e \text{ !! } k \rightarrow^n e'$ then $n \leq (e \text{ !! } k) = (e)(\langle k \rangle) = (e)(k+1)$, i.e. at most $(e)(k+1)$ reduction steps are needed to compute the k^{th} element of a stream e .*

Productive streams are defined in the literature [20] as terms weakly normalizing to infinite lists, which is in our case equivalent to:

Definition 12 (Productive stream). A stream s is productive if for all value $n :: \text{Nat}$, $s \text{ !! } n$ evaluates to a strict value.

Corollary 2. *A closed stream expression admitting an interpretation is productive.*

PROOF. This is a direct application of Corollary 1.

Corollary 3. *Given a function symbol $f :: [\text{Tau}]^k \rightarrow \text{Tau}^l \rightarrow \text{Tau}$ of a program with a polynomial interpretation $(-)$, there is a second order polynomial P such that if $f e_1 \dots e_{k+l} \rightarrow^n v$ then $n \leq P((e_1), \dots, (e_{k+l}))$, for all closed expressions e_1, \dots, e_{k+l} . This polynomial is precisely (f) .*

The following lemma shows that in an interpreted program, the number of evaluated stream elements is bounded by the interpretation.

Lemma 5. *Given a function symbol $f :: [\text{Tau}]^k \rightarrow \text{Tau}^l \rightarrow \text{Tau}$ of a program with interpretation $(-)$, and closed expressions $e_1, \dots, e_l :: \text{Tau}$, $s_1, \dots, s_k :: [\text{Tau}]$, if $f s_1 \dots s_k e_1 \dots e_l \rightarrow! v$ and $\forall n :: \text{Nat}$, $s_i \text{ !! } n \rightarrow! v_i^n$ then for all closed expressions $s'_1 \dots s'_k :: [\text{Tau}]$ satisfying:*

$$\forall n :: \text{Nat} \text{ if } (n) \leq (f s_1 \dots s_k e_1 \dots e_l) + 1 \text{ then } s'_i \text{ !! } n \rightarrow! v_i^n$$

we have:

$$f s'_1 \dots s'_k e_1 \dots e_l \rightarrow! v.$$

PROOF. Let $N = (f s_1 \dots s_k e_1 \dots e_l)$. Since pattern matching on stream arguments has depth 1, the N^{th} element of a stream cannot be evaluated in less than N steps. $f s_1 \dots s_k e_1 \dots e_l$ evaluates in less than N steps, so at most the first N elements of the input stream expressions can be evaluated, so the reduction steps of $f s_1 \dots s_k e_1 \dots e_l$ and $f s'_1 \dots s'_k e_1 \dots e_l$ are exactly the same.

5. Characterizations of polynomial time

In this section, we provide a characterization of polynomial time UOTM computable functions using interpretations. We also provide a partial characterization of Basic Feasible Functionals using the same methodology.

Definition 13. The expressions in a program represent integers, functions, or type-2 functionals in the following sense:

- an expression $\mathbf{e} :: \mathbf{Bin}$ computes an integer n if \mathbf{e} evaluates to a strict value representing the binary encoding of n .
- an expression $\mathbf{e} :: [\mathbf{Bin}]$ computes a function $f : \mathbb{N} \rightarrow \mathbb{N}$ if $\mathbf{e} !! \mathbf{n}$ computes $f(n)$.
- a function symbol \mathbf{f} computes a function $F : (\mathbb{N} \rightarrow \mathbb{N})^k \rightarrow \mathbb{N}^l \rightarrow \mathbb{N}$ if $\mathbf{f} \mathbf{s}_1 \dots \mathbf{s}_k \mathbf{e}_1 \dots \mathbf{e}_l$ computes $F(g_1, \dots, g_k, x_1, \dots, x_l)$ for all expressions $\mathbf{s}_1, \dots, \mathbf{s}_k, \mathbf{e}_1, \dots, \mathbf{e}_l$ of respective types $[\mathbf{Bin}]$ and \mathbf{Bin} computing some functions g_1, \dots, g_k and integers x_1, \dots, x_l .

Theorem 2. *A function $F : (\mathbb{N} \rightarrow \mathbb{N})^k \rightarrow \mathbb{N}^l \rightarrow \mathbb{N}$ is computable in polynomial time by a UOTM if and only if there exists a program \mathbf{f} computing F , of type $[\mathbf{Bin}]^k \rightarrow \mathbf{Bin}^l \rightarrow \mathbf{Bin}$ which admits a polynomial interpretation.*

To prove this theorem, we will demonstrate in Lemma 6 that second order polynomials can be computed by programs having polynomial interpretations. We will then use this result to get completeness in Lemma 7. Soundness (Lemma 9) consists in computing a bound on the number of inputs to read in order to compute an element of the output stream and then to perform the computation by a classical Turing machine.

5.1. Completeness

Lemma 6. *Every second order polynomial can be computed in unary by a polynomial program.*

PROOF. Examples 6 and 7 give polynomial interpretations of unary addition (**plus**) and multiplication (**mult**) on unary integers (**Nat**). Then, we can define a program \mathbf{f} computing the second order polynomial P by $\mathbf{f} \mathbf{y}_1 \dots \mathbf{y}_k \mathbf{x}_1 \dots \mathbf{x}_l = \mathbf{e}$ where \mathbf{e} is the strict implementation of P :

- X_i is implemented by the zero order variable \mathbf{x}_i .
- $Y_i \langle P_1 \rangle$ is computed by $\mathbf{y}_i !! \mathbf{e}_1$, if \mathbf{e}_1 computes P_1 .
- The constant $n \in \mathbb{N}$ is implemented by the corresponding strict value $\mathbf{n} :: \mathbf{Nat}$.
- $P_1 + P_2$ is computed by **plus** $\mathbf{e}_1 \mathbf{e}_2$, if \mathbf{e}_1 and \mathbf{e}_2 compute P_1 and P_2 respectively.
- $P_1 \times P_2$ is computed by **mult** $\mathbf{e}_1 \mathbf{e}_2$, if \mathbf{e}_1 and \mathbf{e}_2 compute P_1 and P_2 respectively.

Since **plus**, **mult** and **!!** have a polynomial interpretation, (\mathbf{e}) is a second order polynomial P_e in $(\mathbf{y}_1), \dots, (\mathbf{y}_k), (\mathbf{x}_1), \dots, (\mathbf{x}_l)$ and we just set $(\mathbf{f}) = P_e + 1$.

Lemma 7 (Completeness). *Every polynomial time UOTM computable function can be computed by a polynomial program.*

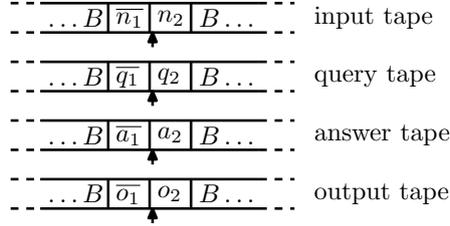


Figure 5: Encoding of the content of the tapes of an OTM (or UOTM). \bar{w} represents the mirror of the word w and the arrows represent the positions of the heads.

PROOF. Let $f : (\mathbb{N} \rightarrow \mathbb{N})^k \rightarrow \mathbb{N}^l \rightarrow \mathbb{N}$ be a function computed by a UOTM \mathcal{M} in time P , with P a second order polynomial. Without loss of generality, we will assume that $k = l = 1$. The idea of this proof is to write a program whose function symbol \mathbf{f}_0 computes the output of \mathcal{M} after t steps, and to use Lemma 6 to simulate the computation of P .

Let \mathbf{f}_0 be the function symbol describing the execution of \mathcal{M} :

$$\mathbf{f}_0 :: [\text{Bin}] \rightarrow \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Bin}^8 \rightarrow \text{Bin}$$

The arguments of \mathbf{f}_0 represent respectively the input stream, the number of computational steps \mathbf{t} , the current state and the 4 tapes (each tape is represented by two binary numbers as illustrated in Figure 5). The output will correspond to the content of the output tape after \mathbf{t} steps.

The function symbol \mathbf{f}_0 is defined recursively in its second argument:

- if the timer is 0, then the program returns the content of the output tape (after its head):

$$\mathbf{f}_0 \ \mathbf{s} \ 0 \ \mathbf{q} \ \mathbf{n}_1 \ \mathbf{n}_2 \ \mathbf{q}_1 \ \mathbf{q}_2 \ \mathbf{a}_1 \ \mathbf{a}_2 \ \mathbf{o}_1 \ \mathbf{o}_2 = \ \mathbf{o}_2$$

- for each transition of \mathcal{M} , we write a definition of this form:

$$\begin{aligned} \mathbf{f}_0 \ \mathbf{s} \ (\mathbf{t}+1) \ \mathbf{q} \ \mathbf{n}_1 \ \mathbf{n}_2 \ \mathbf{q}_1 \ \mathbf{q}_2 \ \mathbf{a}_1 \ \mathbf{a}_2 \ \mathbf{o}_1 \ \mathbf{o}_2 \\ = \ \mathbf{f}_0 \ \mathbf{s} \ \mathbf{t} \ \mathbf{q}' \ \mathbf{n}'_1 \ \mathbf{n}'_2 \ \mathbf{q}'_1 \ \mathbf{q}'_2 \ \mathbf{a}'_1 \ \mathbf{a}'_2 \ \mathbf{o}'_1 \ \mathbf{o}'_2 \end{aligned}$$

where \mathbf{n}_1 and \mathbf{n}_2 represent the input tape before the transition and \mathbf{n}'_1 and \mathbf{n}'_2 represent the input tape after the transition, the motion and writing of the head being taken into account, and so on for the other tapes.

Since the transition function is well described by a set of such definitions, the function \mathbf{f}_0 produces the content of \mathbf{o}_2 (*i.e.* the content of the output tape) after t steps on input \mathbf{t} and configuration \mathcal{C} (*i.e.* the state and the representations of the tapes).

f_0 admits a polynomial interpretation $\langle f_0 \rangle$. Indeed, in each definition, the state can only increase by a constant, the length of the numbers representing the various tapes cannot increase by more than 1. The answer tape $\langle a_2 \rangle$ can undergo an important increase: when querying, it can increase by $\langle s \rangle(\langle q_2 \rangle)$, that is the interpretation of the input stream taken in the interpretation of the query.

Then we can provide a polynomial interpretation to f_0 :

$$\begin{aligned} \langle f_0 \rangle(Y, T, Q, N_1, N_2, Q_1, Q_2, A_1, A_2, O_1, O_2) = \\ (T + 1) \times (Y \langle Q_2 \rangle + 1) + Q + N_1 + N_2 + Q_1 + A_1 + A_2 + O_1 + O_2 \end{aligned}$$

Lemma 6 shows that the polynomial P can be implemented by a program p with a polynomial interpretation. Finally, consider the programs `size`, `max`, `maxsize` and f_1 defined below:

```
size :: Bin -> Nat
size Nil = 0
size (0 x) = (size x)+1
size (1 x) = (size x)+1

max :: Nat -> Nat -> Nat
max 0 0 = 0
max 0 (k+1) = k+1
max (n+1) 0 = n+1
max (n+1) (k+1) = (max n k)+1

maxsize :: [Bin] -> Nat -> Nat
maxsize (h:t) 0 = size h
maxsize (h:t) (n+1) = max (maxsize t n) (size h)

f1 :: [Bin] -> Bin -> Bin
f1 s n = f0 s (p (maxsize s) (size n)) q0 Nil n Nil ... Nil
```

where q_0 is the index of the initial state, `size` computes the size of a binary number, and `maxsize` computes the size function of a stream of binary numbers. f_1 computes an upper bound on the number of steps before \mathcal{M} halts on input n with oracle s (i.e. $P(|s|, |n|)$), and simulates f_0 within this time bound. The output is then the value computed by \mathcal{M} on these inputs. Define the following polynomial interpretations for `max`, `size` and `maxsize`:

$$\begin{aligned} \langle \text{size} \rangle(X) &= 2X \\ \langle \text{max} \rangle(X_1, X_2) &= X_1 + X_2 \\ \langle \text{maxsize} \rangle(Y, X) &= 2 \times Y \langle X \rangle \end{aligned}$$

Finally f_1 admits a polynomial interpretation since it is defined by composition of programs with polynomial interpretations.

Adapting the proof of Lemma 7 in the case of an UOTM without type-1 input (*i.e.* an ordinary Turing machine) allows us to state a similar result for type-1 functions (which can already be deduced from well known results in the literature).

Corollary 4. *Every polynomial time computable type-1 function can be computed by a polynomial stream-free program.*

5.2. Soundness

In order to prove the soundness result, we need to prove that the polynomial interpretation of a program can be computed in polynomial time by a UOTM in this sense:

Lemma 8. *If P is a second-order polynomial, then the function:*

$$F_1, \dots, F_k, x_1, \dots, x_l \mapsto 2^{P(|F_1|, \dots, |F_k|, |x_1|, \dots, |x_l|)} - 1$$

is computable in polynomial time by a UOTM.

PROOF. The addition and multiplication on unary integers, and the function $x \mapsto |x|$ are clearly computable in polynomial time. Polynomial time is also stable under composition, so we only need to prove that the size function $|F|$ is computable in polynomial time by a UOTM. This is the case since it is a max over a polynomial number of elements of polynomial length. Note that this would not be true with the size function $||\cdot||$ as defined in [3] since is not computable in polynomial time.

Lemma 9 (Soundness). *If a function symbol $f :: [\text{Bin}]^k \rightarrow \text{Bin}^l \rightarrow \text{Bin}$ admits a polynomial interpretation, then it computes a type-2 function f of type $(\mathbb{N} \rightarrow \mathbb{N})^k \rightarrow \mathbb{N}^l \rightarrow \mathbb{N}$ which is computable in polynomial time by a UOTM.*

PROOF. From the initial program (which uses streams), we can build a program using finite lists instead of streams as follows.

For each inductive type Tau , let us define the inductive type of finite lists over Tau :

```
data List(Tau) = Cons Tau List(Tau) | []
```

The type of each function symbol is changed from $[\text{Bin}]^k \rightarrow \text{Bin}^l \rightarrow \text{Bin}$ to $\text{List}(\text{Bin})^k \rightarrow \text{Bin}^l \rightarrow \text{Nat} \rightarrow \text{Bin}$ (or from $[\text{Bin}]^k \rightarrow \text{Bin}^l \rightarrow [\text{Bin}]$ to $\text{List}(\text{Bin})^k \rightarrow \text{Bin}^l \rightarrow \text{Nat} \rightarrow \text{List}(\text{Bin})$): streams are replaced by lists, and there is an extra unary argument. We also add an extra constructor **Err** to each type definition.

For each definition in the program, we replace the stream constructor $(:)$ with the list constructor (**Cons**). We also add extra definitions matching the cases where some of the list arguments match the empty list ($[]$). In this case, the left part is set to **Err**. Whenever a function is applied to this special value, it returns it. This defines a new program with only inductive types, which behaves similarly to the original one.

The following program transforms a finite list of binary words into a stream by completing it with zeros.

```

app :: List(Bin) -> [Bin]
app (Cons h t) = h : (app t)
app [] = 0 : (app [])

```

It is easy to verify that $\langle \mathbf{app} \rangle(\mathbf{X})(\mathbf{Z}) = \mathbf{X} + 3\mathbf{Z}$ is a correct interpretation. On strict values, the new program:

$$\mathbf{f} \, \mathbf{v}_1 \ \dots \ \mathbf{v}_k \ \mathbf{v}_{k+1} \ \mathbf{v}_{k+l}$$

reduces as the original one with lists completed with \mathbf{app} :

$$\mathbf{f} \ (\mathbf{app} \ \mathbf{v}_1) \ \dots \ (\mathbf{app} \ \mathbf{v}_k) \ \mathbf{v}_{k+1} \ \mathbf{v}_{k+l}.$$

The only differences are the additional reductions steps for \mathbf{app} , but there is at most one additional reduction step for each stream argument each time there is a definition reduction ((*d*) rule in Figure 4), so the total number of reductions is at most multiplied by a constant. The evaluation may also terminate earlier if the error value appears at some point.

The number of reduction steps is then bounded (up to a multiplicative constant) by:

$$\langle \mathbf{f} \rangle(\langle \mathbf{app} \ \mathbf{v}_1 \rangle, \dots, \langle \mathbf{app} \ \mathbf{v}_k \rangle, \langle \mathbf{v}_{k+1} \rangle, \dots, \langle \mathbf{v}_{k+l} \rangle)$$

Since $\langle \mathbf{app} \ \mathbf{v}_i \rangle$ is a first order polynomial in the interpretation of \mathbf{v}_i , Lemma 1 proves that the previous expression is a first order polynomial in the interpretation of its inputs. Several results in the literature (for example [21, 22]) on type-1 interpretations allow us to state that this new program computes a polynomial type-1 function (in particular because it is an orthogonal term rewriting system).

Let us now build a UOTM which computes \mathbf{f} . According to Lemma 8, given some inputs and oracles, we can compute $\langle \mathbf{f} \rangle$ applied to their sizes and get a unary integer N in polynomial time. The UOTM then computes the first N values of each type-1 input to obtain finite lists (of polynomial size) and then compute the corresponding list function on these inputs. According to Lemma 5, since the input lists are long enough, the result computed by the list function and the stream function are the same.

5.3. Basic feasible functionals

In the completeness proof (Lemma 7), the program built from the UOTM deals with streams using only the $!!$ function to simulate oracle calls. This translation can be easily adapted to implement OTMS with stream programs. Because of the strict inclusion between polynomial time UOTM computable functions and BFF (*cf.* Lemma 2) and the Lemma 9, it will not always be possible to provide a polynomial interpretation to this translation, but a new proof provides us with a natural class of interpretation functions.

Definition 14 (exp-poly). Let exp-poly be the set of functions generated by the following grammar:

$$EP := P \mid EP + EP \mid EP \times EP \mid Y\langle 2^{EP} \rangle$$

The interpretation of a program is exp-poly if each symbol is interpreted by an exp-poly function.

Example 10. $P(Y, X) = Y \langle 2^{X^2+1} \rangle$ is in exp-poly, whereas $2^X \times Y \langle 2^X \rangle$ is not.

Theorem 3. *Every BFF functional is computed by a program which admits an exp-poly interpretation.*

PROOF. Let f be a function computed by an OTM \mathcal{M} . We can reuse the construction from the proof of lemma 7 to generate a function symbol \mathbf{f}_0 (and its definitions) simulating the machine. Since we consider an OTM and no longer a UOTM, we should note that when querying, the query tape now contains a binary word, which we have to convert into unary before giving it to the $!!$ function using an auxiliary function `natofbin`:

```
natofbin :: Bin -> Nat
natofbin Nil = 0
natofbin (0 x) = plus (natofbin x) (natofbin x)
natofbin (1 x) = (plus (natofbin x) (natofbin x)) +1
```

Its interpretation should verify, using the interpretation of `plus` given in Example 6:

$$\langle \text{natofbin} \rangle (X + 1) > 3 \times \langle \text{natofbin} \rangle (X) + 1$$

This can be fulfilled with $\langle \text{natofbin} \rangle (X) = 2^{2^X}$. Note that this conversion function has no implementation with sub-exponential interpretation since the size of its output is exponential in the size of the input.

Now, in the new program, oracle calls are translated into `s !! (natofbin a2)`, and the increase in $\langle \mathbf{a2} \rangle$ is now bounded by $\langle \mathbf{s} \rangle (2^{\langle 2^{\mathbf{a2}} \rangle})$.

We can define the interpretation of \mathbf{f}_0 by:

$$\langle \mathbf{f}_0 \rangle (Y, T, \dots, Q_2, \dots) = (T+1) \times (Y \langle 2^{2^{Q_2}} \rangle + 1) + Q + N_1 + N_2 + Q_1 + A_1 + A_2 + O_1 + O_2$$

Exp-poly functions are also computed by exp-poly programs using the same composition of `natofbin` and $!!$. Finally, the adaptation of the initial proof shows that \mathbf{f}_1 will also have an exp-poly interpretation.

Remark 4. The soundness proof (Lemma 9) does not adapt to BFF and exp-poly programs, because the required analogue of Lemma 8 (where P is an exp-poly and the function is computable by a polynomial time OTM) is false (in particular, $x, F \mapsto 2^{2^{F(x)}}$ is not basic feasible). Still, we conjecture that the converse of Theorem 3 holds. That is exp-poly programs only compute BFF functionals.

6. Link with polynomial time computable real functions

We show in this section that our complexity results can be adapted to real functions.

Until now, we have considered stream programs as type-2 functionals in their own rights. However, type-2 functionals can be used to represent real functions. Indeed Recursive Analysis models computation on reals as computation on converging sequences of rational numbers [23, 17]. Note that there are numerous other possible applications, for example Kapoulas [24] uses UOTM to study the complexity of p -adic functions and the following results could be adapted, since p -adic numbers can be seen as streams of integers between 1 and $p - 1$.

We will require a given convergence speed to be able to compute effectively. A real x is represented by a sequence $(q_n)_{n \in \mathbb{N}} \in \mathbb{Q}^{\mathbb{N}}$ if:

$$\forall i \in \mathbb{N}, |x - q_i| < 2^{-i}.$$

This will be denoted by $(q_n)_{n \in \mathbb{N}} \rightsquigarrow x$. In other words, after a query of size n , an UOTM obtains an encoding of a rational number q_n approaching its input with precision 2^{-n} .

Definition 15 (Computable real function). A function $f : \mathbb{R} \rightarrow \mathbb{R}$ will be said to be computed by an UOTM if:

$$(q_n)_{n \in \mathbb{N}} \rightsquigarrow x \Rightarrow (\mathcal{M}(q_n))_{n \in \mathbb{N}} \rightsquigarrow f(x). \quad (1)$$

We will restrict to functions over the real interval $[0, 1]$ (or any compact set).

Hence a computable real function will be computed by programs of type $[Q] \rightarrow [Q]$ in our stream language, where Q is an inductive type describing the set of rationals \mathbb{Q} . For example, we can define the data type of rationals with a pair constructor $/$:

```
data Q = Bin / Bin
```

Only programs encoding machines verifying the implication (1) will make sense in this framework. Following [17], we can define polynomial complexity of real functions using polynomial time UOTM computable functions.

The following theorems are applications of Theorem 2 to this framework.

Theorem 4 (Soundness). *If a program $f :: [Q] \rightarrow [Q]$ with a polynomial interpretation computes a real function, then this function is computable in polynomial time.*

PROOF. Let us define g from $f :: [Q] \rightarrow [Q]$:

```
g :: [Q] -> Nat -> Q
g s n = (f s) !! n
```

If f has a polynomial interpretation $(\llbracket f \rrbracket)$, then g admits the polynomial interpretation $(\llbracket g \rrbracket)(Y, N) = (\llbracket f \rrbracket)(Y, N) + 1$ (using the usual interpretation of $!!$ defined in Example 8).

The type of rational numbers can be seen as pairs of binary numbers, so Theorem 2 can be easily adapted to this framework. A machine computes a real function in polynomial time if and only if it outputs the n^{th} element of the result

in polynomial with respect to the size of the input (as defined in Definition 4) and n . In this sense, the machine constructed from g using Theorem 2 computes the real function computed by f .

Example 11. The square function over the real interval $[0, 1]$ can be implemented in our stream language, provided we already have a function symbol $\mathbf{bsqr} :: \mathbf{Bin} \rightarrow \mathbf{Bin}$ implementing the square function over binary integers:

```
Rsqr :: [Q] -> [Q]
Rsqr (h : t) = Ssqr t
```

```
Ssqr :: [Q] -> [Q]
Ssqr ((a / b) : t) = (bsqr a / bsqr b) : (Ssqr t)
```

\mathbf{Ssqr} squares each element of its input stream. \mathbf{Rsqr} removes the head of its input before applying \mathbf{Ssqr} because the square function requires one additional precision unit on its input:

$$\forall n \in \mathbb{N}, |q_n - x| \leq 2^{-n} \Rightarrow \forall n \in \mathbb{N}, |q_{n+1}^2 - x^2| \leq 2^{-n}$$

This allows us to prove that the output stream converges at the right speed and that \mathbf{Rsqr} indeed computes the real square function on $[0, 1]$. Now, set the polynomial interpretations:

$$\langle \mathbf{Ssqr} \rangle(Y, Z) = 2 \times Z \times \langle \mathbf{bsqr} \rangle(Y \langle Z \rangle)$$

$$\langle \mathbf{Rsqr} \rangle(Y, Z) = 2 \times Z \times \langle \mathbf{bsqr} \rangle(Y \langle Z + 1 \rangle)$$

They are indeed valid:

$$\begin{aligned} \langle \mathbf{Ssqr} \rangle((a / b) : t) \langle Z + 1 \rangle &= 2(Z + 1) \langle \mathbf{bsqr} \rangle(2 + A + B + T \langle Z \rangle) \\ &> 2 + \langle \mathbf{bsqr} \rangle(A) + \langle \mathbf{bsqr} \rangle(B) + 2Z \langle \mathbf{bsqr} \rangle(T \langle Z - 1 \rangle) \\ &= \langle \mathbf{bsqr} \rangle(a / b) : \langle \mathbf{Ssqr} \rangle(t) \end{aligned}$$

and

$$\begin{aligned} \langle \mathbf{Rsqr} \rangle(h : t) \langle Z \rangle &= 2Z \langle \mathbf{bsqr} \rangle(1 + H + T \langle Z - 1 + 1 \rangle) \\ &> 2Z \langle \mathbf{bsqr} \rangle(T \langle Z \rangle) = \langle \mathbf{Ssqr} \rangle(t) \end{aligned}$$

Theorem 5 (Completeness). *Any polynomial-time computable real function can be implemented by a polynomial program.*

PROOF. Following [17], we can describe any computable real function f by two functions $f_{\mathbb{Q}} : \mathbb{N} \times \mathbb{Q} \rightarrow \mathbb{Q}$ and $f_m : \mathbb{N} \rightarrow \mathbb{N}$ where $f_{\mathbb{Q}}(n, q)$ computes an approximation of $f(q)$ with precision 2^{-n} :

$$\forall q, (f_{\mathbb{Q}}(n, q))_{n \in \mathbb{N}} \rightsquigarrow f(q)$$

and f_m is a modulus of continuity of f defined as follows:

$$\forall n, x, y, |x - y| < 2^{-f_m(n)} \Rightarrow |f(x) - f(y)| < 2^{-n}$$

For a polynomial-time computable real function, those f_m and f_Q are discrete functions computable in polynomial time. Corollary 4 ensures that these functions can be implemented by programs \mathbf{f}_Q and \mathbf{f}_m with polynomial interpretations. Then, we can easily derive a program that computes f by first finding which precision on the input is needed (using f_m) and computing with f_Q an approximation of the image of the input.

```

 $\mathbf{f}_{\text{aux}} :: \text{Nat} \rightarrow [\mathbf{Q}] \rightarrow [\mathbf{Q}]$ 
 $\mathbf{f}_{\text{aux}} \ n \ y = (\mathbf{f}_Q \ n \ (y \ !! \ (\mathbf{f}_m \ n))) : (\mathbf{f}_{\text{aux}} \ (n+1) \ y)$ 
 $\mathbf{f} :: [\mathbf{Q}] \rightarrow [\mathbf{Q}]$ 
 $\mathbf{f} \ y = \mathbf{f}_{\text{aux}} \ 0 \ y$ 

```

with \mathbf{Q} an inductive type representing rational numbers.

We can easily check that these interpretations work:

$$(\mathbf{f}_{\text{aux}})(Y, N, Z) = (Z + 1) \times (1 + (\mathbf{f}_Q)(N + Z, Y \langle (\mathbf{f}_m)(N + Z) \rangle))$$

$$(\mathbf{f})(Y, Z) = 1 + (\mathbf{f}_{\text{aux}})(Y, 1, Z)$$

7. Conclusion

We have provided a characterization of polynomial time stream complexity using basic polynomial interpretations and this the first characterization of this kind. More complex and finer interpretation techniques on first order complexity classes (*e.g.* sup-interpretations or quasi-interpretations) could probably be adapted to stream languages.

This work also provides a partial characterization of BFF and shows that it is not the right feasible complexity class for functions over streams. Our framework also adapts well to applications like computable analysis. We have indeed characterized the class of polynomial time real functions, and this is again the first time that this class is characterized using interpretations.

As a whole, this work is a first step toward higher order complexity. We have used second order interpretations, but higher order interpretations could also be used to characterize higher order complexity classes. The main difficulty is that the state of the art on higher order complexity rarely deal with orders higher than two.

References

- [1] R. L. Constable, Type two computational complexity, in: Proc. 5th annual ACM STOC, 108–121, 1973.
- [2] K. Mehlhorn, Polynomial and abstract subrecursive classes, in: Proceedings of the sixth annual ACM symposium on Theory of computing, ACM New York, NY, USA, 96–109, 1974.

- [3] B. M. Kapron, S. A. Cook, A new characterization of type-2 feasibility, *SIAM Journal on Computing* 25 (1) (1996) 117–132.
- [4] A. Seth, Turing machine characterizations of feasible functionals of all finite types, *Feasible Mathematics II* (1995) 407–428.
- [5] R. J. Irwin, J. S. Royer, B. M. Kapron, On characterizations of the basic feasible functionals (Part I), *J. Funct. Program.* 11 (1) (2001) 117–153.
- [6] R. Ramyaa, D. Leivant, Ramified Corecurrence and Logspace, *Electronic Notes in Theoretical Computer Science* 276 (0) (2011) 247 – 261, ISSN 1571-0661.
- [7] R. Ramyaa, D. Leivant, Feasible Functions over Co-inductive Data, in: *Logic, Language, Information and Computation*, vol. 6188 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, ISBN 978-3-642-13823-2, 191–203, 2010.
- [8] S. Bellantoni, S. A. Cook, A New Recursion-Theoretic Characterization of the Polytime Functions, *Computational Complexity* 2 (1992) 97–110.
- [9] D. Leivant, J.-Y. Marion, Lambda Calculus Characterizations of Poly-Time, *Fundam. Inform.* 19 (1/2) (1993) 167–184.
- [10] G. Bonfante, J.-Y. Marion, J.-Y. Moyén, Quasi-interpretations a way to control resources, *Theoretical Computer Science* 412 (25) (2011) 2776 – 2796, ISSN 0304-3975.
- [11] P. Baillot, U. D. Lago, Higher-Order Interpretations and Program Complexity, in: P. Cégielski, A. Durand (Eds.), *CSL*, vol. 16 of *LIPICs*, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, ISBN 978-3-939897-42-2, 62–76, 2012.
- [12] Z. Manna, S. Ness, On the termination of Markov algorithms, in: *Third Hawaii international conference on system science*, 789–792, 1970.
- [13] D. Lankford, On proving term rewriting systems are Noetherian, *Tech. Rep.*, Mathematical Department, Louisiana Technical University, Ruston, Louisiana, 1979.
- [14] J.-Y. Marion, R. Péchoux, Sup-interpretations, a semantic method for static analysis of program resources, *ACM Trans. Comput. Logic* 10 (4) (2009) 27:1–27:31, ISSN 1529-3785.
- [15] M. Gaboardi, R. Péchoux, Upper Bounds on Stream I/O Using Semantic Interpretations, in: E. Grädel, R. Kahle (Eds.), *CSL*, vol. 5771 of *Lecture Notes in Computer Science*, Springer, ISBN 978-3-642-04026-9, 271–286, 2009.
- [16] R. M. Amadio, Synthesis of max-plus quasi-interpretations, *Fundamenta Informaticae* 65 (1) (2005) 29–60.

- [17] K.-I. Ko, Complexity theory of real functions, Birkhauser Boston Inc. Cambridge, MA, USA, 1991.
- [18] H. Férée, E. Hainry, M. Hoyrup, R. Péchoux, Interpretation of Stream Programs: Characterizing Type 2 Polynomial Time Complexity, in: Algorithms and Computation, vol. 6506 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, ISBN 978-3-642-17516-9, 291–303, 2010.
- [19] R. Péchoux, Synthesis of sup-interpretations: A survey, *Theor. Comput. Sci.* 467 (2013) 30–52.
- [20] J. Endrullis, C. Grabmayer, D. Hendriks, A. Ishihara, J. W. Klop, Productivity of stream definitions, *Theor. Comput. Sci.* 411 (4-5) (2010) 765–782.
- [21] G. Bonfante, A. Cichon, J.-Y. Marion, H. Touzet, Algorithms with polynomial interpretation termination proof, *Journal of Functional Programming* 11 (01) (2001) 33–53.
- [22] U. Lago, S. Martini, Derivational Complexity Is an Invariant Cost Model, in: M. Eekelen, O. Shkaravska (Eds.), *Foundational and Practical Aspects of Resource Analysis*, vol. 6324 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, ISBN 978-3-642-15330-3, 100–113, 2010.
- [23] K. Weihrauch, *Computable analysis: an introduction*, Springer Verlag, 2000.
- [24] G. Kapoulas, Polynomially Time Computable Functions over p-Adic Fields, in: J. Blanck, V. Brattka, P. Hertling (Eds.), *Computability and Complexity in Analysis*, vol. 2064 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, ISBN 978-3-540-42197-9, 101–118, 2001.

5 On Bounding Space Usage of Streams Using Interpretation Analysis

On Bounding Space Usage of Streams Using Interpretation Analysis

Marco Gaboardi, Romain Péchoux

► **To cite this version:**

Marco Gaboardi, Romain Péchoux. On Bounding Space Usage of Streams Using Interpretation Analysis. Science of Computer Programming, Elsevier, 2015, pp.44. hal-01112161

HAL Id: hal-01112161

<https://hal.inria.fr/hal-01112161>

Submitted on 2 Apr 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



On Bounding Space Usage of Streams Using Interpretation Analysis

Marco Gaboardi^{1,2}

Harvard University and University of Dundee

Romain Péchoux¹

Université de Lorraine - INRIA project Carte, Loria, UMR 7503

Abstract

Interpretation methods are important tools in implicit computational complexity. They have been proved particularly useful to statically analyze and to limit the complexity of programs. However, most of these studies have been so far applied in the context of term rewriting systems over finite data.

In this paper, we show how interpretations can also be used to study properties of lazy first-order functional programs over streams. In particular, we provide some interpretation criteria useful to ensure two kinds of stream properties: *space upper bounds* and *input/output upper bounds*. Our space upper bounds criteria ensures global and local upper bounds on the size of each output stream element expressed in terms of the maximal size of the input stream elements. The input/output upper bounds criteria consider instead the relations between the number of elements read from the input stream and the number of elements produced on the output stream.

This contribution can be seen as a first step in the development of a methodology aiming at using interpretation properties to ensure space safety properties of programs working on streams.

Keywords: Stream Programs, Interpretations, Program Space Usage, Lazy Languages, Implicit Computational Complexity.

Email addresses: m.gaboardi@dundee.ac.uk (Marco Gaboardi), pechoux@loria.fr (Romain Péchoux)

URL: <http://www.cs.unibo.it/~gaboardi> (Marco Gaboardi),
<http://www.loria.fr/~pechoux> (Romain Péchoux)

¹Work partially supported by the projects ANR-14-CE25-0005 “ELICA”, MIUR-PRIN’07 “CONCERTO” and INRIA Associated Team “CRISTAL”.

²The research leading to these results has received funding from the European Community’s Seventh Framework Programme (FP7/2007-2013) under grant agreement n° 272487.

1. Introduction

The advances obtained in communication technology in the last two decades have posed new challenges to the software community. One of these challenges comes from the advancements achieved in computer networking where new software able to handle huge amount of data in an efficient way is required.

This situation has brought a renewed interest for stream-like data structures and for programs managing those data structures. Indeed, by representing discrete potentially infinite information flows, streams can be used to formalize and study situations as real-time data processing, network communication flows, audio and video signals flows, etc. Clearly, the problems that stream programs raise are different from the ones generally considered in the usual scenario where data are assumed to be finite. For this reason, several programming languages have been proposed with the aim of modeling stream-based computations, see [39] for a survey.

The aim of the present work is to contribute to the current scenario by developing some static analysis techniques useful to ensure basic properties of lazy functional programs working on streams. This is the first step of a more general investigation of complexity and efficiency properties of programs working on streams.

Stream-like languages and properties. Several formal frameworks have been designed for the manipulation of infinite objects including infinitary rewriting [23] and infinitary lambda-calculus [24]. Important properties of these models such as infinitary weak normalization and infinitary strong normalization have been deeply studied in the literature. However, little attention has been paid to space properties of such models. A different setting handling infinite data-structures is computable analysis, which provides several models of computation over real numbers [40]. In this setting a lot of work has been done to adapt the classical concept of complexity class and obtain implicit characterizations. However, even if streams can be considered as particular real numbers, the properties of interest for stream programs are usually different from the ones of interest in computable analysis.

A well-established approach to deal with infinite data, and in particular with streams, is by using *laziness* in functional programming languages [21]. In languages like Haskell, streams are expressions denoting infinite lists whose elements are evaluated on demand. In this way streams can be treated by finitary means. The practical diffusion of lazy programming languages has stimulated the development of tools and techniques in order to prove properties of programs in the presence of infinite data structures.

For example, on the side of program equivalence much attention has been paid to the study of co-induction and bisimulation techniques in languages working on streams [35, 20]. A property of stream definitions that has motivated many studies is productivity [14]. A stream definition is productive if it can be effectively evaluated in a unique constructor infinite normal form. Productivity is in general undecidable, so, many restricted languages and restricted criteria have been studied to ensure it [38, 12, 22, 15].

Besides program equivalence and productivity, other stream program properties, in particular space-related properties, have received little attention. Such properties are studied in this paper through the use of interpretations, a static analysis tool.

Interpretations. Interpretation methods originate from the natural observation that, in order to reason about program properties, it is sometimes more convenient to interpret syntactic program constructions into the objects of an abstract domain and prove properties about the obtained abstract objects.

Interpretation methods have been proved useful in many situations and are nowadays well-established verification tools for proving properties of programs. Variants of interpretation have been used for example to prove the termination of term rewriting systems [31, 26], to obtain sound approximations of program behaviors useful to static analysis [13] and to obtain implicit characterizations of complexity classes [7, 32].

One variation of particular interest for implicit computational complexity is the notion of quasi-interpretation [7]. A quasi-interpretation maps program constructions to functions over real numbers. The mapping is chosen in such a way that the function obtained as the interpretation of a program describes an upper bound on the size of the computed values with respect to the size of input values. Thanks to this, quasi-interpretations are particularly adapted to study program complexity in an elegant way.

Another important property of quasi-interpretation is that the problem of finding a quasi-interpretation of a given program for some restricted class of polynomials is decidable [2, 9]. This suggests that quasi-interpretations can be used as a concrete tool to analyze the complexity of functional programs.

An important new issue is whether interpretations can be used in order to infer such properties on programs computing over infinite data. Here we approach this problem by considering lazy programs over stream data.

Contribution. In this paper, we consider a simple first-order lazy language and we start a systematic study of *space properties* of programs working on streams by means of interpretation methods.

In many stream applications one is interested in processing data in a fast and memory-safe way. In order to do this, one can think to improve space-efficiency by using some buffering operations to memorize only the part of the stream involved in the actual computation. Following this intuition, it becomes natural to study space properties of programs working on streams in a more abstract way. We study two classes of space properties:

- *Stream Upper Bounds*: these are properties about the size of each stream element produced by a program. They correspond to properties about the elements memorized in the buffer.
- *Bounded Input/Output Properties*: these are properties about the number of stream elements produced by a program. They correspond to properties about the number of elements produced on the output wrt to the number of elements read on the input.

These properties analyze two “dimensions” of programs working on streams. The combination of these properties allows one to study a reasonable class of programs and to obtain the information needed in order to improve the memory management process of programs working on streams.

The results presented in this paper have been originally developed in [18] and [19]. In [19], we mainly studied the space upper bounds properties while in [18] we studied the bounded input/output properties. The present paper generalizes and extends these works, in particular, to a pure functional programming style. Indeed previous works were restricted to term rewrite systems and the adaptation of interpretation methods to pure functional programs is a new non-trivial feature. Consequently, new proofs but also more illustrating diagrams and examples have been provided. Finally, a deeper comparison with the state of the art on stream properties (productivity, complexity, ...) and related works has been provided in Section 7.

Stream Upper Bounds. In order to process stream data in a memory-efficient way it is useful to obtain an estimate of the memory needed to store the elements produced by a stream program.

In some situations, an estimate can be obtained by considering in a *global* way the greatest size of the elements produced by the program as outputs. In other situations, however, there is no such a maximal element with respect to the size measure and so only an estimate considering the *local* position of the element in the stream can be given. Consider the following stream definitions:

$$\begin{array}{ll} \text{ones} :: [\text{Nat}] & \text{nats} :: \text{Nat} \rightarrow [\text{Nat}] \\ \text{ones} \doteq \underline{1} : \text{ones} & \text{nats } x \doteq x : (\text{nats } (x + 1)) \end{array}$$

In both cases, it is easy to obtain such estimates. Indeed, in the stream definition of `ones` all the elements have the same size, while in the definition of `nats` every element has a size depending on its position in the stream.

However, when more complex stream programs are considered, deeper analyses are needed. In this paper, we will use interpretations to define two criteria useful to compute both kinds of space estimates.

Consider the following stream program:

$$\begin{array}{ll} \text{repeat} :: \text{Nat} \rightarrow [\text{Nat}] & \text{zip} :: [a] \rightarrow [a] \rightarrow [a] \\ \text{repeat } x \doteq x : (\text{repeat } x) & \text{zip } (x : xs) ys \doteq x : (\text{zip } ys xs) \end{array}$$

It is easy to verify that the size of every element of a stream `s` built only using `repeat` and `zip` is bounded by a constant n , i.e. the maximal natural number encoding \underline{n} in a subterm `repeat` \underline{n} in `s`. In particular, it means that every stream `s` built only using `repeat` and `zip` is globally bounded by a constant n . In order to generalize this informal analysis, we study a *Global Upper Bound* (GUB) criterion ensuring that the size of stream elements is bounded by a function in the maximal size of the input elements.

Analogously, consider the following stream program:

$$\begin{array}{ll} \text{nats} :: \text{Nat} \rightarrow [\text{Nat}] & \text{sadd} :: [\text{Nat}] \rightarrow [\text{Nat}] \rightarrow [\text{Nat}] \\ \text{nats } x \doteq x : (\text{nats } (x + 1)) & \text{sadd } (x : xs) (y : ys) \doteq (\text{add } x y) : (\text{sadd } xs ys) \end{array}$$

Every stream `s` built using `nats` and `sadd` is not globally bound. Nevertheless it is easy for every such an `s` to compute a function f such that every element of `s` in the *local* position n has a size bounded by $f(n)$. In order to generalize this informal argument, we study a *Local Upper Bound* (LUB) criterion ensuring that the size of the n -th eval-

uated element of a stream is bounded by a function in its index n and the maximal size of the input. All the productive stream functions have a local upper bound, however in order to establish a criteria ensuring it we need an extension of the usual notion of interpretation. For this reason we introduce the notion of *parametrized interpretation*, i.e. an interpretation where functions depend on external parameters.

Bounded Input/Output Properties. Another information that is useful to obtain in order to improve memory-efficiency is an estimate of the number of elements produced by a stream program when fed with only a portion of the input stream. Indeed if one think to online streaming, these properties consist in bounding the speed-up that might occur in the network during communication.

In some situations, such an estimate can be obtained by considering only the *length* of the portion of the input stream. In other situations, however, this is not sufficient and so in order to obtain the estimate one needs to consider also the *size* of the elements in the portion. Consider the following definitions:

$$\begin{aligned} \text{merge} &:: [a] \rightarrow [a] \rightarrow [a \times a] \\ \text{merge } (x : xs) (y : ys) &\doteq (x, y) : (\text{merge } ys xs) \\ \text{dup} &:: [a] \rightarrow [a] \\ \text{dup } (x : xs) &\doteq x : (x : (\text{dup } xs)) \end{aligned}$$

It is easy to verify that each stream expression built using only `merge` and `dup` will only generate a number of output elements that depends on the number of input read elements; e.g. the expression `dup (merge (dup s) (dup s))` for each read element of the input stream `s` produces four elements of the type $a \times a$. In order to generalize this informal argument, we study a *Length-Based Upper Bound* (LBUB) criterion ensuring that the number m of output stream elements is bounded by a function in the number n of stream elements in input.

Many stream functions have a length-based upper bound. However, there are stream functions that generate a number of output elements that does not depend only on the number of input read elements. Consider the following definitions:

$$\begin{aligned} \text{app} &:: [a] \rightarrow [a] \rightarrow [a] & \text{upto} &:: \text{Nat} \rightarrow [\text{Nat}] \\ \text{app } (x : xs) ys &\doteq x : (\text{app } xs ys) & \text{upto } 0 &\doteq \text{nil} \\ \text{app nil } ys &\doteq ys & \text{upto } (x + 1) &\doteq (x + 1) : (\text{upto } x) \\ \text{extendupto} &:: [\text{Nat}] \rightarrow [\text{Nat}] \\ \text{extendupto } (x : xs) &\doteq \text{app } (\text{upto } x) (\text{extendupto } xs) \end{aligned}$$

It is easy to verify that every stream expression built using only `upto` and `extendupto` will generate a number of output elements that is related to both the number and the size of input read elements; e.g. the expression: `extendupto (extendupto s)` for each natural number \underline{n} in the input stream `s` outputs $\sum_{i=1}^{\underline{n}} i$ elements. In order to generalize this informal argument, we study a *Size-Based Upper Bound* (SBUB) criterion ensuring that the number m of output stream elements is bounded by a function in the number and the size of the stream elements in input.

Other Technical Contributions. Besides the study of stream program properties, this paper contains two other technical contributions:

- a definition of interpretations for a lazy first-order programming language
- a definition of a new kind of interpretations, named *parametrized interpretations*

Interpretations have been so far presented as tools dealing with properties about rewriting systems. Here instead, we are interested in programs of a first-order lazy functional program. A possible approach could have been to translate programs in a term rewriting system and analyze them using the standard interpretation framework. Instead, we have adapted the interpretation tools to our case. This choice is due on the one hand to the desire to have a treatment as close as possible to the programming language, on the other hand this is also due to the desire of understanding the flexibility and the adaptability of the interpretation tools.

Parametrized interpretations extend standard interpretations by means of an external parameter. In the parametrized interpretations, all the functions appearing in the assignments can depend on external parameters. However, the parameter has a different status with respect to the other arguments of the functions. Thanks to this extension, we are able to deal with properties about stream local positions as required by the Local Upper Bound property.

Outline of the paper. In Section 2, we introduce the language, named SFL, and some notations. In Section 3, we introduce interpretations and parametrized interpretations. In Section 4, we study the space upper bound properties and the semantic interpretation criteria to ensure them. In Section 5, we consider the bounded input/output upper bound properties and how to ensure them through interpretation criteria. In section 6, we discuss the problem of computing program interpretations. In Section 7, we present the related works. In Section 8, we draw some conclusions.

2. The SFL language

In the present section, we introduce the syntax and the operational semantics of the language that will be used all along this paper. The language is dubbed SFL, acronym for *Stream First-order Lazy language*. This is an Haskell-like lazy first-order language computing on simple *stream* data.

We consider programs of SFL to be well-typed closed expressions of base type. Programs can be evaluated thanks to a lazy big step semantics where by *lazy* we mean that the evaluation does not go under a constructor. This permits to deal with streams and infinite computations in a natural way. Indeed, analogously to what happens in Haskell, we can prove program properties by equational reasoning. However, our operational semantics differs from the Haskell one since we do not consider sharing.

2.1. Syntax and Types

Let \mathcal{X} , \mathcal{C} and \mathcal{F} be three disjoint sets representing the set of *variables*, the set of *constructor symbols* (or constructors) and the set of *function symbols* respectively. In the sequel, x , c , f and t denote symbols in \mathcal{X} , \mathcal{C} , \mathcal{F} and $\mathcal{C} \cup \mathcal{F}$, respectively.

Definition 1. *The syntax of the SFL language is described by the following grammar:*

$p ::= x \mid c(x, \dots, x)$	(Patterns)
$e ::= x \mid c(e, \dots, e) \mid f(e, \dots, e) \mid \text{LetRec } d \text{ in } e \mid$ $\text{Case } e \text{ of } p \rightarrow e, \dots, p \rightarrow e$	(Expressions)
$d ::= f(x, \dots, x) \doteq e$	(Definitions)
$v ::= c(e, \dots, e)$	(Lazy Values)
$\underline{v} ::= c(\underline{v}, \dots, \underline{v})$	(Strict Values)

In the examples presented in the sequel, the set of constructor symbols will include the usual constructors for natural numbers (i.e. $0, + 1$), lists (i.e. $\text{nil}, :$) and pairs (i.e. $\langle \cdot, \cdot \rangle$) and it may also include other standard algebraic data types. Besides, we assume the set of constructors to contain also a constructor Err that will be used to track pattern matching failures.

We consider *patterns* that are either a variable or a constructor possibly applied to other variables. For simplicity, we assume that a variable can appear at most once in a pattern and that the patterns are non-overlapping.

Expressions can be built using variables, constructors, function symbols, the LetRec construction and the Case construction. We consider a grammar where constructors and functions symbols do not appear partially applied in an expression, e.g a function symbol f of arity two will only appear in the form $f(e_1, e_2)$, for some expressions e_1 and e_2 .

The Case constructor as usual allows one to perform pattern matching. Note that even if the patterns are built by using (at most) one constructor at a time, by using nested Case more complex patterns can be explored. The LetRec construction is used to locally define recursive functions. In particular, a construction like $\text{LetRec } d_f \text{ in } e$ has two parameters: a *function definition* d_f and an expression e . The function definition d_f is the actual place where a recursive definition is assigned to the function symbol f . The expression e is the scope of that definition.

We distinguish two kinds of *values*: *lazy* and *strict values*. The semantics in the next subsection evaluates programs to lazy values. Strict values are specific lazy values that will be used to define the program analyses presented in the following sections. In particular, later in this section, we will show how to define an eval program forcing the evaluation of a program to a strict value.

Free and *bound* variables are defined as usual. However, free variables in expressions can be also explicitly bound in *definitions*, that is: given a definition d of the shape $f(x_1, \dots, x_n) \doteq e$, the bound variables of d are the ones of e and x_1, \dots, x_n . Note also that a LetRec construction can bind function symbols. That is, the function symbol f is bound in e' in an expression of the shape $\text{LetRec } f(x_1, \dots, x_n) \doteq e \text{ in } e'$. For simplicity, we assume that all the bound variables and function symbols have distinct names so that name clashes are avoided.

As outlined above, we are mainly concerned with stream programs properties related to space. So, we need to introduce a notion of *size* for expressions and programs.

Definition 2 (Size). *The size of an expression e , denoted $|e|$, is defined as*

- $|x| = 1$

- $|\tau(e_1, \dots, e_n)| = \begin{cases} 0 & \text{if } \tau \text{ is of arity } 0 \\ \sum_{1 \leq i \leq n} |e_i| + 1 & \text{if } \tau \text{ is of arity } n \geq 1 \end{cases}$
- $|\text{LetRec } d \text{ in } e| = |e|$
- $|\text{Case } e \text{ of } p_1 \rightarrow e_1, \dots, p_n \rightarrow e_n| = |e| + \max_{1 \leq i \leq n} |e_i|$

Note that if we take \mathbb{N} to be the set of natural number expressions inductively defined using the constructors 0 and $+ 1$ then for each $\underline{n} \in \mathbb{N}$ we have $|\underline{n}| = n$, i.e. the size of a natural number expression is equal to the value it represents.

In the sequel only expressions that are well-formed and well-typed will be considered.

Definition 3.

- A definition $\mathfrak{f}(x_1, \dots, x_n) \doteq e$ is well-formed if and only if all the free variables of e are among $x_1 \dots x_n$, i.e. the definition has no free variables.
- An expression e is well-formed if and only if for every function symbol \mathfrak{f} there is exactly one well-formed function definition d defining it, i.e. of the shape $\mathfrak{f}(x_1, \dots, x_n) \doteq e'$.

We conclude this part by describing some of the notations we will use in the sequel. In presenting the examples we adopt the standard applicative convention for the parenthesis (as in Haskell), e.g. we use $\mathfrak{f}(x + 1) 0$ to denote $\mathfrak{f}(x + 1, 0)$. We use the vector notation \vec{e} as a shorthand for a sequence of expressions as e_1, \dots, e_n . So, for instance the expression $\tau(e_1, \dots, e_n)$ could be also written as $\tau \vec{e}$. Finally, given a sequence of expressions \vec{e} and a function F on expressions, we use $F(\vec{e})$ to denote $F(e_1), \dots, F(e_n)$, i.e. the componentwise application of F to the sequence \vec{e} . For instance, given a sequence $\vec{e} = e_1, \dots, e_n$, we use $|\vec{e}|$ as a notation for $|e_1|, \dots, |e_n|$.

Type system. As stressed before, we want to consider only expressions that are well-typed. Here we introduce the type system that assigns types to all the syntactic constructions of the SFL language. As usual, the type system ensures that a program does not go wrong. Roughly speaking, a *wrong computation* happens when a program cannot be evaluated to a value because of some stuck computation. Note however that this does not prevent a program from either diverging or evaluating to `Err`. Indeed, in our setting, this fact is important both for making the pattern matching working properly and also for using some program analysis techniques presented in the following sections.

In order to make simpler our analyses, we only consider well-typed first-order programs dealing with lists that do not contain other lists. This is because we want to prevent object like streams of streams that cannot be analyzed in a proper way by the methods that we will present in the sequel. We assure this property by a typing restriction similar to the one of [17]. The following type definition reflects this and the fact that we restrict our attention only to first-order programs.

Definition 4. *The SFL types are defined by the following grammar:*

$\frac{\Gamma(\mathbf{x}) = \mathbf{A}}{\Gamma; \Delta \vdash \mathbf{x} :: \mathbf{A}} \text{ (Var)}$	$\frac{}{\Gamma; \Delta \vdash \mathbf{Err} :: \mathbf{A}} \text{ (A - Err)}$
$\frac{\mathbf{c} :: \mathbf{A}_1 \rightarrow \dots \rightarrow \mathbf{A}_n \rightarrow \mathbf{A} \quad \Gamma; \Delta \vdash \mathbf{e}_i :: \mathbf{A}_i \ (1 \leq i \leq n)}{\Gamma; \Delta \vdash \mathbf{c}(\mathbf{e}_1, \dots, \mathbf{e}_n) :: \mathbf{A}} \text{ (Con)}$	
$\frac{\Delta(\mathbf{f}) = \mathbf{A}_1 \rightarrow \dots \rightarrow \mathbf{A}_n \rightarrow \mathbf{A} \quad \Gamma; \Delta \vdash \mathbf{e}_i :: \mathbf{A}_i \ (1 \leq i \leq n)}{\Gamma; \Delta \vdash \mathbf{f}(\mathbf{e}_1, \dots, \mathbf{e}_n) :: \mathbf{A}} \text{ (Fun)}$	
$\frac{\Gamma; \Delta \vdash \mathbf{e} :: \mathbf{A} \quad \Gamma_i; \emptyset \vdash \mathbf{p}_i :: \mathbf{A} \quad \Gamma, \Gamma_i; \Delta \vdash \mathbf{e}_i :: \mathbf{B} \ (1 \leq i \leq m)}{\Gamma; \Delta \vdash \mathbf{Case} \ \mathbf{e} \ \mathbf{of} \ \mathbf{p}_1 \rightarrow \mathbf{e}_1, \dots, \mathbf{p}_m \rightarrow \mathbf{e}_m :: \mathbf{B}} \text{ (Case)}$	
$\frac{\Gamma, \mathbf{x}_1 :: \mathbf{A}_1, \dots, \mathbf{x}_n :: \mathbf{A}_n, ; \Delta, \mathbf{f} : \mathbf{A}_1 \rightarrow \dots \rightarrow \mathbf{A}_n \rightarrow \mathbf{A} \vdash \mathbf{e} :: \mathbf{A}}{\Gamma; \Delta \vdash \mathbf{f}(\mathbf{x}_1, \dots, \mathbf{x}_n) \doteq \mathbf{e} :: \mathbf{A}_1 \rightarrow \dots \rightarrow \mathbf{A}_n \rightarrow \mathbf{A}} \text{ (Def)}$	
$\frac{\Gamma; \Delta \vdash \mathbf{f}(\mathbf{x}_1, \dots, \mathbf{x}_n) \doteq \mathbf{e} :: \phi \quad \Gamma; \Delta, \mathbf{f} :: \phi \vdash \mathbf{e}_1 :: \mathbf{A}}{\Gamma; \Delta \vdash \mathbf{LetRec} \ \mathbf{f}(\mathbf{x}_1, \dots, \mathbf{x}_n) \doteq \mathbf{e} \ \mathbf{in} \ \mathbf{e}_1 :: \mathbf{A}} \text{ (Letrec)}$	

Table 1: SFL type system

σ	$::=$	$a \mid \mathbf{Nat} \mid \sigma \times \sigma$	<i>(basic types)</i>
\mathbf{A}	$::=$	$\alpha \mid \sigma \mid \mathbf{A} \times \mathbf{A} \mid [\sigma]$	<i>(base types)</i>
ϕ	$::=$	$\mathbf{A} \mid \mathbf{A} \rightarrow \phi$	<i>(types)</i>

where a is a basic type variable, α is a type variable, \mathbf{Nat} is a constant type representing natural numbers, \times and $[\]$ are base type constructors for pairs and (finite and infinite lists) streams respectively.

As stressed above, it is worth noticing that the above definition can be extended to other algebraic data types. In the sequel, we use a, b to denote basic type variables, α, β to denote type variables, σ, τ for basic data types, \mathbf{A}, \mathbf{B} to denote base types and ϕ for types. We will tacitly use restricted polymorphism, i.e. a basic type variable a and a type variable α will represent every basic and base type respectively.

For notational convenience, we will use the vector notation $\vec{\mathbf{A}} \rightarrow \mathbf{B}$ as an abbreviation for $\mathbf{A}_1 \rightarrow \dots \rightarrow \mathbf{A}_n \rightarrow \mathbf{B}$.

The type system proves two kinds of typing judgments: $\Gamma; \Delta \vdash \mathbf{e} :: \mathbf{A}$ for expressions, and $\Gamma; \Delta \vdash \mathbf{f}(\mathbf{x}_1, \dots, \mathbf{x}_n) \doteq \mathbf{e} :: \phi$ for function definitions. In particular, the judgments for expressions assign a *base type* to an expression, while the judgments for function definitions assign a *type* to a function definition. The symbols Γ and Δ denote variables and function symbols *contexts* respectively; that is, partial functions assigning types to variables and function symbols respectively. Note that we do not consider constants symbols in contexts but instead we assume that they come with a fixed type signature.

Definition 5. *Well-typed expressions and function definitions are defined using the type*

system in Table 1.

It is worth noticing that the symbol Err can be typed with each base type A . This is essential in order to get type preservation in the evaluation mechanism. Note also that the functional types can be assigned to constructor and function symbols, but only base types can be assigned to expressions. Consequently, our language only allows programs with first-order function definitions.

Definition 6. A SFL program is a well-formed expression e that is typable through a type judgment of the shape $\emptyset; \emptyset \vdash e :: A$ such that A does not contain free type variables.

While the above definition could seem a bit odd, it is easy to verify that this corresponds to the usual notion of programs considered as closed terms of observable types.

Notations for the examples. The language we introduced above makes the interpretation definitions we will provide in the following section more formal. In contrast, concrete examples can be cumbersome. So, in the remainder of the paper we will use some *syntactic sugar* to improve readability. Let us start to show that we can use general forms of pattern matching in our examples. Note that we have introduced patterns following the grammar:

$$p ::= x \mid c(x, \dots, x)$$

So, in particular we do not have patterns for nested constructors. However, more complex pattern matching can be easily simulated through the use of combined Case constructions. As an example consider a function f that we want to define by pattern matching on expressions of the shape $(x + 1) + 1$. This can be defined as follow:

$$f \ y_1 = \text{Case } y_1 \text{ of } y_2 + 1 \rightarrow \text{Case } y_2 \text{ of } x + 1 \rightarrow e$$

Instead of writing this in full form, we will simply write it as:

$$f \ ((x + 1) + 1) \doteq e$$

More generally, we will use the notation:

$$\begin{aligned} f \ p_1^1 \cdots p_n^1 &\doteq e_1 \\ &\vdots \\ f \ p_1^k \cdots p_n^k &\doteq e_k \end{aligned}$$

as syntactic sugar for a function definition of the shape:

$$\begin{aligned} f(x_1, \dots, x_n) &\doteq \text{Case } x_1 \text{ of } p_1^1 \rightarrow \dots \text{Case } x_n \text{ of } p_n^1 \rightarrow e_1 \\ &\vdots \\ & p_1^k \rightarrow \dots \text{Case } x_n \text{ of } p_n^k \rightarrow e_k \end{aligned}$$

More complex examples consisting of function definitions with several distinct function symbols will be treated analogously by juxtaposition of their syntactic sugars. We

then assume these definitions to be bound by a `LetRec` for some particular expression under consideration. That is, we will usually consider an expression e in isolation but this has to be considered as a program of the shape:

$$\text{LetRec } d_1, \dots, d_n \text{ in } e$$

where d_1, \dots, d_n are all the function definitions for the function symbols in e .

Stream terminology. The program analysis methods we will present in the following sections are specific to the study of stream program properties. This means that we will pay particular attention to programs working on the type $[A]$, the type of both finite and infinite lists of type A .

We distinguish two classes of functions symbols useful to work with streams. Following the terminology of [15], we have:

Definition 7.

- A function symbol f is a stream function if $f :: [\sigma_1] \rightarrow \dots \rightarrow [\sigma_n] \rightarrow \vec{\tau} \rightarrow [\sigma]$, with $n > 0$. Conversely, a function symbol f is a stream constructor if $f :: \vec{\tau} \rightarrow [\sigma]$.
- A function definition d_f such that $f(x_1, \dots, x_n) = e$ is a stream function definition if f is a stream function. Conversely, if f is a stream constructor we say that d_f is a stream definition.

Intuitively, we call stream functions those functions that transform and combine input streams to produce an output stream. Analogously, we call stream constructors those functions that can be used to actually produce new output streams from scratch.

Example 1. Consider the following definitions:

<code>odd :: [a] → [a]</code>	<code>nats :: Nat → [Nat]</code>
<code>odd (x : y : xs) ≐ x : (odd xs)</code>	<code>nats x ≐ x : (nats (x + 1))</code>
<code>zip :: [a] → [a] → [a]</code>	<code>nodd :: [Nat]</code>
<code>zip (x : xs) ys ≐ x : (zip ys xs)</code>	<code>nodd ≐ odd (nats (0 + 1))</code>

We have that `odd` and `zip` are two stream functions of one and two arguments respectively, while both `nats` and `nodd` are stream constructors. Note that the fact of being a stream constructor does not impose limitations on the kind of functions that can be used in the right-hand side of the definition. Indeed, in the `nodd` example, we use both a stream function (i.e. `odd`) and a stream constructor (i.e. `nats`).

2.2. Lazy operational semantics

In this section, we describe the SFL operational semantics. As outlined above, the operational semantics can be described by means of a *lazy* big-step semantics. With the term *lazy*, in the tradition of [33, 1], we identify a semantics that does not evaluate

$\frac{c \in \mathcal{C}}{\mathcal{H}; c(\mathbf{e}_1, \dots, \mathbf{e}_n) \Downarrow c(\mathbf{e}_1, \dots, \mathbf{e}_n)} \text{ (val)}$	$\frac{\mathcal{H} \cup \{\mathbf{d}\}; \mathbf{M} \Downarrow \mathbf{v}}{\mathcal{H}; \text{LetRec } \mathbf{d} \text{ in } \mathbf{M} \Downarrow \mathbf{v}} \text{ (rec)}$
$\frac{\mathcal{H}; \mathbf{e}\{\mathbf{e}_1/\mathbf{x}_1, \dots, \mathbf{e}_n/\mathbf{x}_n\} \Downarrow \mathbf{v} \quad (\mathbf{f} \ \mathbf{x}_1 \ \dots \ \mathbf{x}_n \doteq \mathbf{e}) \in \mathcal{H}}{\mathcal{H}; \mathbf{f}(\mathbf{e}_1, \dots, \mathbf{e}_n) \Downarrow \mathbf{v}} \text{ (fun)}$	
$\frac{\mathcal{H}; \mathbf{e} \Downarrow c(\mathbf{e}'_1, \dots, \mathbf{e}'_m) \quad \mathbf{p}_i = c(\mathbf{x}_1, \dots, \mathbf{x}_m) \quad \mathcal{H}; \mathbf{e}_i\{\mathbf{e}'_1/\mathbf{x}_1, \dots, \mathbf{e}'_m/\mathbf{x}_m\} \Downarrow \mathbf{v}}{\mathcal{H}; \text{Case } \mathbf{e} \text{ of } \mathbf{p}_1 \rightarrow \mathbf{e}_1, \dots, \mathbf{p}_n \rightarrow \mathbf{e}_n \Downarrow \mathbf{v}} \text{ (pm)}$	
$\frac{\mathcal{H}; \mathbf{e} \Downarrow c(\mathbf{e}'_1, \dots, \mathbf{e}'_m) \quad \forall i \leq n, \mathbf{p}_i \neq c(\mathbf{x}_1, \dots, \mathbf{x}_m)}{\mathcal{H}; \text{Case } \mathbf{e} \text{ of } \mathbf{p}_1 \rightarrow \mathbf{e}_1, \dots, \mathbf{p}_n \rightarrow \mathbf{e}_n \Downarrow \text{Err}} \text{ (pm}_e\text{)}$	

Table 2: SFL lazy operational semantics

under the constructors. So in particular, we do not consider the sharing issue that is studied in other lazy and call-by-need semantics [27, 3].

In order to describe the semantics, we need two additional components: *substitutions* and *environments*. A *substitution* $\{\mathbf{e}_1/\mathbf{x}_1, \dots, \mathbf{e}_n/\mathbf{x}_n\}$ is a partial function mapping variables to expressions. As usual we denote $\mathbf{e}\{\mathbf{e}_1/\mathbf{x}_1, \dots, \mathbf{e}_n/\mathbf{x}_n\}$ the result of the application of the substitution $\{\mathbf{e}_1/\mathbf{x}_1, \dots, \mathbf{e}_n/\mathbf{x}_n\}$ to the free variables of \mathbf{e} . An *environment* is simply a set of well-formed function definitions. We will use the letter \mathcal{H} to denote environments.

Definition 8. *The operational evaluation relation \Downarrow is the relation between environments, expressions and lazy values inductively defined by the rules in Table 2.*

Intuitively, the judgment $\mathcal{H}; \mathbf{e} \Downarrow \mathbf{v}$ means that the expression \mathbf{e} can be evaluated to the lazy value \mathbf{v} using the rules of the semantics and the function definitions contained in the environment \mathcal{H} . For notational convenience, we simply write $\mathbf{e} \Downarrow \mathbf{v}$ for $\mathcal{H}; \mathbf{e} \Downarrow \mathbf{v}$ when $\mathcal{H} = \emptyset$. Moreover, in the sequel when we write $\mathcal{H}; \mathbf{e} \Downarrow \mathbf{v}$ we implicitly assume that \mathcal{H} contains the function definitions for all the function symbols in \mathbf{e} .

It is worth noticing that as usual in lazy semantics the abstract machine does not explore the entire result but stops once the requested information is found; this is why the axiom rule (*val*) only refers to lazy values. Moreover, as anticipated in the previous subsection, we use the constructor `Err` to deal with pattern matching errors. This should not be confused with the errors that can be generated by programs that go wrong. Indeed, to prevent such situations types are sufficient, as usual.

Strict evaluation. We have introduced the operational semantics of our language SFL in the previous paragraph. We have defined it to be lazy since our main concern is to deal with infinite computations in a natural way. However, another concern of our work is to describe program analysis techniques using only finitary operational tools without making reference to infinite abstract domains. For this reason, sometimes we will need to consider the complete evaluation of values. This is why we have introduced the category of *strict values* in the grammar definition in Definition 1.

In order to evaluate programs to strict values, we can define a particular function eval_A for every base type A as follows:

$$\begin{aligned} \text{eval}_A &:: A \rightarrow A \\ \text{eval}_A \quad (c \ x_1 \ \dots \ x_n) &\doteq \hat{C}(\text{eval}_{A_1} \ x_1) \ \dots \ (\text{eval}_{A_n} \ x_n) \end{aligned}$$

where \hat{C} is a function symbol representing the *strict* version of the primitive constructor c . For instance in the case where c is $+ 1$ we can define \hat{C} to be the function $\text{succ} :: \text{Nat} \rightarrow \text{Nat}$ defined as:

$$\begin{aligned} \text{succ} \quad \text{Err} &\doteq \text{Err} + 1 \\ \text{succ} \quad 0 &\doteq 0 + 1 \\ \text{succ} \quad (x + 1) &\doteq (x + 1) + 1 \end{aligned}$$

When we want to stress that an expression e is completely evaluated (i.e. that besides being an expression, it is also a strict value) we use the notation \underline{e} . A relevant set of completely evaluated expressions is the set N of *canonical numerals* defined as:

$$N = \{ \underline{n} \mid \underline{n} = \underbrace{((\dots (0 + 1) \dots) + 1)}_{n \text{ times}} \text{ and } \underline{n} :: \text{Nat} \}$$

A concrete example of computation by strict evaluation can be found in Appendix B.

More notations. For notational convenience, in the sequel we will use the notation:

$$\mathcal{H}, e \Downarrow_v \underline{v}$$

to denote the judgment:

$$\mathcal{H}, \text{eval}_A \ e \Downarrow \underline{v}$$

assuming that the type A is made clear by the context. Moreover, we introduce some notation for some well-established functions that we will use in the following sections.

We use the notation $e_{\underline{n}}$ as a shorthand for the expression $e \ !! \ \underline{n}$ where $!!$ is the usual indexing function returning the n -th element of a list. That is:

$$\begin{aligned} !! &:: [a] \rightarrow \text{Nat} \rightarrow a \\ \text{nil} \quad !! \quad y &\doteq \text{Err} \\ (x : \text{xs}) \quad !! \quad 0 &\doteq x \\ (x : \text{xs}) \quad !! \quad (y + 1) &\doteq \text{xs} \ !! \ y \end{aligned}$$

We use the shorthand $e|_{\underline{n}}$ to denote the expression $\text{take } \underline{n} \ e$ where take is the usual function which returns the first n elements of a list:

$$\begin{aligned} \text{take} &:: \text{Nat} \rightarrow [a] \rightarrow [a] \\ \text{take} \quad 0 \quad s &\doteq \text{nil} \\ \text{take} \quad (x + 1) \quad \text{nil} &\doteq \text{Err} \\ \text{take} \quad (x + 1) \quad (y : \text{ys}) &\doteq y : (\text{take } x \ \text{ys}) \end{aligned}$$

Finally, we use lg to denote the function that returns the number of elements in a finite partial list:

$$\begin{aligned}
\text{lg} &:: [a] \rightarrow \text{Nat} \\
\text{lg} \quad \text{nil} &\doteq \underline{0} \\
\text{lg} \quad \text{Err} &\doteq \underline{0} \\
\text{lg} \quad (x : xs) &\doteq (\text{lg } xs) + 1
\end{aligned}$$

In the sequel, we tacitly assume that the above definitions are contained in all the environments \mathcal{H} that we will consider.

3. Interpretation

The program analyses that we will introduce in Sections 4 and 5 will be based on the notion of *interpretation*. Intuitively, an interpretation consists of an assignment mapping each symbol of a program to a function over non-negative real numbers. Thanks to the real numbers ordering, such a peculiar assignment combined with some additional criteria permits to prove program properties.

This kind of reasoning is inspired by the notion of *polynomial interpretation* [31, 26, 6], developed in the field of program termination, and by the notions of *quasi-interpretation* [8] and *sup-interpretation* [32], developed more recently in the field of implicit computational complexity.

We now stress the main distinctions between the notion of interpretation presented in this section and the standard notion of interpretations on Term Rewrite Systems (see the survey [7]). In Subsections 3.1 and 3.2, we define the notions of *assignment* and *interpretation*. These definitions are similar to the one on TRS ([7]). The only distinction is that these notions are adapted to the presented functional language (the case construct is treated). The notions of *additive* and *monotonic assignments* are also standard. The only new notion is the notion of *almost-additive* (see Definition 3) allowing to deal with stream construct in a more flexible manner. All the results relating the size of a value and its interpretation (e.g. Corollary 1) or the interpretations of a term and its evaluation (e.g. Proposition 1) are fairly standard so an expert reader may go directly to Subsection 3.3 where a new notion of *parametrized interpretation* is defined. This notion will be useful for the *Local Upper Bound* (LUB) criterion.

3.1. Assignment

In the following, an *assignment* is used as a method to map in a canonical way programs to non-negative real numbers (i.e. elements of \mathbb{R}^+) in such a way that a comparison of programs is possible thanks to the usual ordering on real numbers. In order to do this, an assignment maps program components either to non-negative real numbers or to functions over non-negative real numbers.

Definition 9 (Assignment).

- A variable assignment, denoted ρ is a map associating to each $x \in \mathcal{X}$ a value r in \mathbb{R}^+ .
- A symbol assignment, denoted ξ is a map associating to each symbol $\mathfrak{t} \in \mathcal{C} \cup \mathcal{F}$ a function $F : \mathbb{R}^+ \times \dots \times \mathbb{R}^+ \rightarrow \mathbb{R}^+$ of the same arity.

- Given a variable assignment ρ and a symbol assignment ξ , an assignment is the extension of ρ and ξ to expressions defined as follows:

$$\begin{aligned}
& - \langle \mathbf{Err} \rangle_{\rho, \xi} = 0 \\
& - \langle \mathbf{x} \rangle_{\rho, \xi} = \rho(x) \\
& - \langle \mathbf{t}(e_1, \dots, e_n) \rangle_{\rho, \xi} = \xi(\mathbf{t})(\langle e_1 \rangle_{\rho, \xi}, \dots, \langle e_n \rangle_{\rho, \xi}) \\
& - \langle \mathbf{LetRec\ d\ in\ e} \rangle_{\rho, \xi} = \langle e \rangle_{\rho, \xi} \\
& - \langle \mathbf{Case\ e\ of\ } c_1(\vec{x}_1) \rightarrow e_1, \dots, c_m(\vec{x}_m) \rightarrow e_m \rangle_{\rho, \xi} \\
& \quad = \max_{1 \leq i \leq m} \{ \langle e_i \rangle_{\rho\{\vec{x}_i = \vec{r}_i\}, \xi} \mid \vec{r}_i \in \mathbb{R}^+ \text{ and } \langle e \rangle_{\rho, \xi} \geq \langle c_i(\vec{x}_i) \rangle_{\rho\{\vec{x}_i = \vec{r}_i\}, \xi} \}
\end{aligned}$$

We consider variable and symbol assignments as total functions over program variables and program symbols, respectively. We write $\vec{r} \in \mathbb{R}^+$ as a shorthand for $\forall r \in \vec{r}, r \in \mathbb{R}^+$, and we write $\rho\{x := r\}$ for the variable assignment defined as ρ except for the variable x to which it assigns the value r . We often abbreviate $\rho\{x_1 := r_1\} \cdots \{x_n := r_n\}$ by $\rho\{\vec{x} = \vec{r}\}$ (as for instance in the definition above). Note that we consider the constructor **Err** differently from the other constructors. This because as we will see later we want that interpretations behave well with respect to pattern matching.

The definition of assignment for the **Case** construction requires the existence of a maximal element $\langle e_i \rangle_{\rho\{\vec{x}_i = \vec{r}_i\}, \xi}$ for $1 \leq i \leq m$ and for \vec{r} ranging over values in \mathbb{R}^+ . The existence of such an element (or equivalently a bound on the search space) is ensured by the side condition $\langle e \rangle_{\rho, \xi} \geq \langle p_i \rangle_{\rho\{\vec{x}_i = \vec{r}_i\}, \xi}$ and by the fact that e does not contain the variables \vec{x} .

Example 2. Consider the following function definitions:

```

add :: Nat → Nat → Nat
add 0 y ≐ y
add (z + 1) y ≐ (add z y) + 1

```

```

sadd :: [Nat] → [Nat] → [Nat]
sadd (x' : xs) (y' : ys) ≐ (add x' y') : (sadd xs ys)

```

Concretely, in SFL they can be computed by the environment \mathcal{H} including the following definitions:

```

add x y ≐ Case x of 0 → y, z + 1 → (add z y) + 1
sadd x y ≐ Case x of x' : xs → Case y of y' : ys → (add x' y') : (sadd xs ys)

```

For each variable assignment $\rho = \{x := r, y := s\}$ and symbol assignment ξ such that $\xi(0) = 0$, $\xi(\cdot)(X, Y) = X + Y + 1$, $\xi(+1)(X) = X + 1$ and $\xi(\mathbf{add})(X, Y) = \xi(\mathbf{sadd})(X, Y) = X + Y$, we compute the assignment of the expression $\mathbf{add\ x\ y}$ as follows:

$$\begin{aligned}
\langle \mathbf{add\ x\ y} \rangle_{\rho, \xi} &= \xi(\mathbf{add})(\langle \mathbf{x} \rangle_{\rho, \xi}, \langle \mathbf{y} \rangle_{\rho, \xi}) \\
&= \rho(x) + \rho(y) \\
&= r + s
\end{aligned}$$

We compute the assignment of Case x of $0 \rightarrow y, z + 1 \rightarrow (\text{add } z \ y) + 1$ in a similar way:

$$\begin{aligned}
& \llbracket \text{Case } x \text{ of } 0 \rightarrow y, z + 1 \rightarrow (\text{add } z \ y) + 1 \rrbracket_{\rho, \xi} \\
&= \max(\max\{\llbracket y \rrbracket_{\rho, \xi} \mid \llbracket x \rrbracket_{\rho, \xi} \geq \llbracket 0 \rrbracket_{\rho, \xi}\}, \\
&\quad \max\{\llbracket (\text{add } z \ y) + 1 \rrbracket_{\rho\{z:=t\}, \xi} \mid t \in \mathbb{R}^+ \text{ and } \llbracket x \rrbracket_{\rho, \xi} \geq \llbracket z + 1 \rrbracket_{\rho\{z:=t\}, \xi}\}) \\
&= \max(\rho(y), \max\{\rho(y) + \rho\{z := t\}(z) + 1 \mid t \in \mathbb{R}^+ \text{ and } \rho(x) \geq \rho\{z := t\}(z) + 1\}) \\
&= \max(s, \max_{r \geq t+1} \{s + t + 1\}) \quad \text{as } \rho(x) = r \text{ and } \rho(y) = s \\
&= \max(s, s + r) = r + s
\end{aligned}$$

The usual notion of assignment used in the context of interpretations does not distinguish between variable and symbol assignments. In our context, we prefer to keep this distinction because it highlights the extension of assignments to the Case construction and because, as we will see later, an interpretation will fix only the symbol assignments.

The following property shows that assignments internalize the substitution mechanism.

Lemma 1 (Assignment Substitution). *Given an assignment $\llbracket - \rrbracket_{\rho, \xi}$ and an expression $\Gamma, x :: A; \Delta \vdash e :: B$, for every expression $\Gamma; \Delta \vdash e' :: A$ we have:*

$$\llbracket e\{e'/x\} \rrbracket_{\rho, \xi} = \llbracket e \rrbracket_{\rho\{x:=\llbracket e' \rrbracket_{\rho, \xi}\}, \xi}$$

Proof. By induction on the structure of e . In the case where e is the variable x then the conclusion follows immediately. The cases where e is either **Err** or a variable distinct from x are trivial. The case where e is a **LetRec** follows directly by induction hypothesis.

Consider now the case $e = \tau(e_1, \dots, e_n)$, by definition we have $\llbracket e\{e'/x\} \rrbracket_{\rho, \xi} = \xi(\tau)(\llbracket e_1\{e'/x\} \rrbracket_{\rho, \xi}, \dots, \llbracket e_n\{e'/x\} \rrbracket_{\rho, \xi})$. By induction hypothesis, $\llbracket e_i\{e'/x\} \rrbracket_{\rho, \xi} = \llbracket e_i \rrbracket_{\rho\{x:=\llbracket e' \rrbracket_{\rho, \xi}\}, \xi}$ for each $1 \leq i \leq n$. So, we can conclude:

$$\begin{aligned}
\llbracket e\{e'/x\} \rrbracket_{\rho, \xi} &= \xi(\tau)(\llbracket e_1\{e'/x\} \rrbracket_{\rho, \xi}, \dots, \llbracket e_n\{e'/x\} \rrbracket_{\rho, \xi}) = \\
&\quad \xi(\tau)(\llbracket e_1 \rrbracket_{\rho\{x:=\llbracket e' \rrbracket_{\rho, \xi}\}, \xi}, \dots, \llbracket e_n \rrbracket_{\rho\{x:=\llbracket e' \rrbracket_{\rho, \xi}\}, \xi}) = \llbracket e \rrbracket_{\rho\{x:=\llbracket e' \rrbracket_{\rho, \xi}\}, \xi}
\end{aligned}$$

Consider the case $e = \text{Case } e'' \text{ of } c_1(\vec{y}_1) \rightarrow e_1, \dots, c_n(\vec{y}_n) \rightarrow e_n$, then by definition, for $x \notin \vec{y}_1, \dots, \vec{y}_n$ (because x is supposed to be free in e), we have:

$$\llbracket e\{e'/x\} \rrbracket_{\rho, \xi} = \max_{1 \leq i \leq m} \{\llbracket e_i\{e'/x\} \rrbracket_{\rho\{\vec{y}_i:=\vec{r}_i\}, \xi} \mid \llbracket e''\{e'/x\} \rrbracket_{\rho, \xi} \geq \llbracket c_i(\vec{y}_i) \rrbracket_{\rho\{\vec{y}_i:=\vec{r}_i\}, \xi}\}$$

By induction hypothesis, we obtain:

$$\begin{aligned}
& \{\llbracket e_i\{e'/x\} \rrbracket_{\rho\{\vec{y}_i:=\vec{r}_i\}, \xi} \mid \llbracket e''\{e'/x\} \rrbracket_{\rho, \xi} \geq \llbracket c_i(\vec{y}_i)\{e'/x\} \rrbracket_{\rho\{\vec{y}_i:=\vec{r}_i\}, \xi}\} = \\
& \{\llbracket e_i \rrbracket_{\rho\{\vec{y}_i:=\vec{r}_i, x:=\llbracket e' \rrbracket_{\rho, \xi}\}, \xi} \mid \llbracket e'' \rrbracket_{\rho\{x:=\llbracket e' \rrbracket_{\rho, \xi}\}, \xi} \geq \llbracket c_i(\vec{y}_i) \rrbracket_{\rho\{\vec{y}_i:=\vec{r}_i, x:=\llbracket e' \rrbracket_{\rho, \xi}\}, \xi}\}
\end{aligned}$$

and so we can conclude $\llbracket e\{e'/x\} \rrbracket_{\rho, \xi} = \llbracket e \rrbracket_{\rho\{x:=\llbracket e' \rrbracket_{\rho, \xi}\}, \xi}$. \square

In this paper we will only deal with assignments that are *monotonic* where the monotonicity condition is defined as follows.

Definition 10 (Monotonic Assignment).

- A symbol assignment ξ is monotonic if for any $\mathfrak{t} \in \mathcal{C} \cup \mathcal{F}$, $\xi(\mathfrak{t})$ is a monotonic function, i.e. $\forall r, s \in \mathbb{R}^+$ s.t. $r \geq s$:

$$\xi(\mathfrak{t})(\dots, r, \dots) \geq \xi(\mathfrak{t})(\dots, s, \dots)$$

- An assignment $\llbracket - \rrbracket_{\rho, \xi}$ is monotonic if the symbol assignment ξ is monotonic.

Notice that the above definition of monotonicity concerns only the constants and the function symbols. This does not imply that all the functions used in an assignment are monotonic. In particular, the Case construction can be interpreted as a function $\llbracket \text{Case } e \text{ of } p_1 \rightarrow e_1, \dots, p_m \rightarrow e_m \rrbracket_{\rho, \xi}$ that is monotonic in $\llbracket e \rrbracket_{\rho, \xi}$ and $\llbracket e_i \rrbracket_{\rho, \xi}$ but not in $\llbracket p_i \rrbracket_{\rho, \xi}$.

Other classes of assignments that will be useful in the sequel are the class of *almost-additive* and *additive* assignments.

Definition 11 (Almost-additive and Additive Assignment).

- The symbol assignment ξ is almost-additive if $\forall c \in \mathcal{C}$ of arity n but the stream constructor $:$, we have:

$$\xi(c)(r_1, \dots, r_n) = \sum_{i=1}^n r_i + \alpha_c, \text{ for some constant } \alpha_c \geq 1, \text{ whenever } n > 0.$$

$$\xi(c) = 0, \text{ otherwise.}$$

The symbol assignment ξ is additive if it is almost-additive and

$$\xi(:)(r_1, r_2) = r_1 + r_2 + \alpha, \text{ for some constant } \alpha \geq 1$$

- An assignment $\llbracket - \rrbracket_{\rho, \xi}$ is an almost-additive (resp. additive) assignment if the symbol assignment ξ is almost-additive (resp. additive).

The fact that an assignment is *additive* is useful in order to relate the interpretation of a strict value to its size. In particular, the following lemma shows that they are linearly related.

Lemma 2. Given an additive assignment $\llbracket - \rrbracket_{\rho, \xi}$, there is a constant α such that for each strict value $\vdash \underline{v} :: \mathbf{A}$ we have:

$$|\underline{v}| \leq \llbracket \underline{v} \rrbracket_{\rho, \xi} \leq \alpha \times |\underline{v}|$$

Proof. We consider $\alpha = \max_{c \in \mathcal{C}} \alpha_c$ and we prove the lemma by induction on the structure of \underline{v} .

In the case \underline{v} is a constructor c of arity 0, by definition we have $|c| = 0 = \llbracket c \rrbracket_{\rho, \xi}$, so the conclusion follows trivially.

Consider now the case $\underline{v} = c(\underline{v}_1, \dots, \underline{v}_n)$. By induction hypothesis for $1 \leq i \leq n$ we have:

$$|\underline{v}_i| \leq (\underline{v}_i)_{\rho, \xi} \leq \alpha \times |\underline{v}_i|$$

So, since by definition we also have:

$$|\underline{v}| = \left(\sum_{1 \leq i \leq n} |\underline{v}_i| \right) + 1 \quad \text{and} \quad (\underline{v})_{\rho, \xi} = \left(\sum_{1 \leq i \leq n} (\underline{v}_i)_{\rho, \xi} \right) + \alpha_c$$

and since $\alpha \geq \alpha_c \geq 1$, using induction hypothesis we can conclude:

$$\sum_{1 \leq i \leq n} |\underline{v}_i| + 1 \leq \sum_{1 \leq i \leq n} (\underline{v}_i)_{\rho, \xi} + \alpha_c \leq \sum_{1 \leq i \leq n} (\alpha \times |\underline{v}_i|) + \alpha_c \leq \alpha \times \left(\sum_{1 \leq i \leq n} |\underline{v}_i| + 1 \right)$$

□

A similar result can be obtained for almost-additive assignments if we restrict the attention to values that are not streams.

Corollary 1. *Given an assignment $(-)_{\rho, \xi}$ such that the symbol assignment ξ is almost-additive, there is a constant α such that for every strict value $\vdash \underline{v} :: \sigma$ we have:*

$$|\underline{v}| \leq (\underline{v})_{\rho, \xi} \leq \alpha \times |\underline{v}|$$

3.2. Interpretations

Now, we are ready to define the main tool that will be used in the next sections: *interpretations*.

Definition 12 (Interpretation). *An expression $\Gamma; \Delta \vdash e :: A$ admits an interpretation $(-)_{\xi}$ if for each variable assignment ρ , the assignment $(-)_{\rho, \xi}$ is monotonic and such that for each function definition $\mathbf{f}(x_1, \dots, x_n) \doteq e'$ the following holds:*

$$(\mathbf{f}(x_1, \dots, x_n))_{\rho, \xi} \geq (e')_{\rho, \xi}$$

The quantification on all variable assignments allows us to reason in general terms about values assigned to variables. For this reason, in the sequel we will usually write X, Y, Z, \dots to denote variables ranging over real numbers; e.g. we will write $(\mathbf{f})_{\xi}(X_1, \dots, X_n)$ for $(\mathbf{f}(x_1, \dots, x_n))_{\xi}$. Analogously, we will write $(e)_{\xi} \geq (e')_{\xi}$ as a shorthands for: $\forall \rho, (e)_{\rho, \xi} \geq (e')_{\rho, \xi}$.

In the sequel, we will need the following substitution property for interpretations.

Lemma 3 (Interpretation Substitution). *Let $\Gamma; \Delta \vdash e :: A$ and $\Gamma, x :: A; \Delta \vdash e_1, e_2 :: B$ be expressions admitting the interpretation $(-)_{\xi}$ and such that $(e_1)_{\xi} \geq (e_2)_{\xi}$. Then:*

$$(e_1 \{e/x\})_{\xi} \geq (e_2 \{e/x\})_{\xi}$$

Proof. By definition of interpretation we have $\forall \rho, (e_1)_{\rho, \xi} \geq (e_2)_{\rho, \xi}$. Thanks to the quantification over all the variable assignments we also have $\forall \rho, (e_1)_{\rho \{x := (e)_{\rho, \xi}\}, \xi} \geq (e_2)_{\rho \{x := (e)_{\rho, \xi}\}, \xi}$. So, by applying Lemma 1, we can conclude $\forall \rho, (e_1 \{e/x\})_{\rho, \xi} \geq (e_2 \{e/x\})_{\rho, \xi}$. □

The inequality conditions required by the definition of interpretation can be naturally inherited by the results of an evaluation. In order to show this we need to extend interpretations to environments.

Definition 13. An environment \mathcal{H} admits the interpretation $\langle \! \langle - \! \rangle \! \rangle_\xi$ if for every variable assignment ρ and every function definition $\mathbf{f}(x_1, \dots, x_n) = \mathbf{e}$ in it, the following holds:

$$\langle \! \langle \mathbf{f}(x_1, \dots, x_n) \! \rangle \! \rangle_{\rho, \xi} \geq \langle \! \langle \mathbf{e} \! \rangle \! \rangle_{\rho, \xi}$$

We can now show some examples.

Example 3. Consider again the environment \mathcal{H} of Example 2. This environment admits an interpretation $\langle \! \langle - \! \rangle \! \rangle_\xi$ if the assignment $\langle \! \langle - \! \rangle \! \rangle_\xi$ satisfies the following inequalities:

$$\begin{aligned} \langle \! \langle \text{add } x \ y \! \rangle \! \rangle_\xi &\geq \langle \! \langle \text{Case } x \text{ of } 0 \rightarrow y, z + 1 \rightarrow (\text{add } z \ y) + 1 \! \rangle \! \rangle_\xi \\ \langle \! \langle \text{sadd } x \ y \! \rangle \! \rangle_\xi &\geq \langle \! \langle \text{Case } x \text{ of } x' : xs \rightarrow \text{Case } y \text{ of } y' : ys \rightarrow (\text{add } x' \ y') : (\text{sadd } xs \ ys) \! \rangle \! \rangle_{\rho\{x':=v, xs:=x\}, \xi} \end{aligned}$$

Consider an additive assignment $\langle \! \langle - \! \rangle \! \rangle_\xi$ such that $\langle \! \langle 0 \! \rangle \! \rangle_{\rho, \xi} = 0$, $\langle \! \langle \cdot \! \rangle \! \rangle_{\rho, \xi}(X, Y) = X + Y + 1$ and $\langle \! \langle +1 \! \rangle \! \rangle_{\rho, \xi}(X) = X + 1$. Now, we are interested in finding an interpretation for the symbols **add** and **sadd** such that $\rho = \{x := r, y := s\}$ satisfies the following inequalities:

$$\begin{aligned} \langle \! \langle \text{add} \! \rangle \! \rangle_{\rho, \xi}(r, s) &\geq \max(\max\{s \mid r \geq 0\}, \max\{1 + \langle \! \langle \text{add} \! \rangle \! \rangle_{\rho, \xi}(t, s) \mid t \in \mathbb{R}^+ \text{ and } r \geq t + 1\}) \\ \langle \! \langle \text{sadd} \! \rangle \! \rangle_{\rho, \xi}(r, s) &\geq \max\{\langle \! \langle \text{Case } y \text{ of } y' : ys \rightarrow (\text{add } x' \ y') : (\text{sadd } xs \ ys) \! \rangle \! \rangle_{\rho\{x':=v, xs:=x\}, \xi} \\ &\quad \mid x, v \in \mathbb{R}^+ \text{ and } r \geq v + x + 1\} \end{aligned}$$

The above can be reformulated as follows:

$$\begin{aligned} \langle \! \langle \text{add} \! \rangle \! \rangle_{\rho, \xi}(r, s) &\geq \max_{\{t \in \mathbb{R}^+ \mid r \geq t + 1\}} (s, 1 + \langle \! \langle \text{add} \! \rangle \! \rangle_{\rho, \xi}(t, s)) \\ \langle \! \langle \text{sadd} \! \rangle \! \rangle_{\rho, \xi}(r, s) &\geq \max\{\langle \! \langle (\text{add } x' \ y') : (\text{sadd } xs \ ys) \! \rangle \! \rangle_{\rho\{x':=v, xs:=x, y':=w, ys:=y\}, \xi} \\ &\quad \mid x, y, v, w \in \mathbb{R}^+ \text{ and } r \geq v + x + 1 \text{ and } s \geq w + y + 1\} \\ &\geq \max\{\langle \! \langle \text{add} \! \rangle \! \rangle_{\rho, \xi}(v, w) + \langle \! \langle \text{sadd} \! \rangle \! \rangle_{\rho, \xi}(x, y) + 1 \\ &\quad \mid x, y, v, w \in \mathbb{R}^+ \text{ and } r \geq v + x + 1 \text{ and } s \geq w + y + 1\} \end{aligned}$$

So, for instance we can choose $\xi(\text{add})(X, Y) = \xi(\text{sadd})(X, Y) = X + Y$ (the first inequality is indeed proved to be an equality in Example 2). With this function symbol assignment ξ , since it is monotonic, we have that $\langle \! \langle - \! \rangle \! \rangle_\xi$ is an interpretation.

Example 4. Consider the function symbol eval_A for strict evaluation. The environment \mathcal{H} that contains definitions of the shape:

$$\text{eval}_A \ x \doteq \text{Case } x \text{ of } (c \ x_1 \ \dots \ x_n) \rightarrow \hat{C}(\text{eval}_{A_1} \ x_1) \ \dots \ (\text{eval}_{A_n} \ x_n)$$

will admit the interpretation $\langle \! \langle - \! \rangle \! \rangle_\xi$ defined by $\xi(c)(X_1, \dots, X_n) = \xi(\hat{C})(X_1, \dots, X_n) = \sum_{i=1}^n X_i + \alpha_c$ and $\forall A, \xi(\text{eval}_A)(X) = X$.

Indeed, consider the assignment $\langle \! \langle - \! \rangle \! \rangle_{\rho, \xi}$ for the environment ρ such that $\rho(x) = r$. On

the one hand, we have $\llbracket \text{eval}_A(\mathbf{x}) \rrbracket_{\rho, \xi} = \xi(\text{eval}_A)((\mathbf{x})_{\rho, \xi}) = \llbracket \mathbf{x} \rrbracket_{\rho, \xi} = \rho(\mathbf{x}) = r$. On the other hand, setting $\vec{x} = x_1, \dots, x_n$ and $\vec{r} = r_1, \dots, r_n$:

$$\begin{aligned}
& \llbracket \text{Case } \mathbf{x} \text{ of } c \mathbf{x}_1 \cdots \mathbf{x}_n \rightarrow \hat{C}(\text{eval}_{A_1} \mathbf{x}_1) \cdots (\text{eval}_{A_n} \mathbf{x}_n) \rrbracket_{\rho, \xi} \\
&= \max\{\llbracket \hat{C}(\text{eval}_{A_1} \mathbf{x}_1) \cdots (\text{eval}_{A_n} \mathbf{x}_n) \rrbracket_{\rho\{\vec{x}=\vec{r}\}, \xi} \mid \llbracket \mathbf{x} \rrbracket_{\rho, \xi} \geq \llbracket c \mathbf{x}_1 \cdots \mathbf{x}_n \rrbracket_{\rho\{\vec{x}=\vec{r}\}, \xi}\} \\
&= \max\{\xi(\hat{C})(\llbracket \text{eval}_{A_1} \mathbf{x}_1 \rrbracket_{\rho\{\vec{x}=\vec{r}\}, \xi}, \dots, \llbracket \text{eval}_{A_n} \mathbf{x}_n \rrbracket_{\rho\{\vec{x}=\vec{r}\}, \xi}) \\
&\quad \mid r \geq \xi(c)(\llbracket \mathbf{x}_1 \rrbracket_{\rho\{\vec{x}=\vec{r}\}, \xi}, \dots, \llbracket \mathbf{x}_n \rrbracket_{\rho\{\vec{x}=\vec{r}\}, \xi})\} \\
&= \max\{\sum_{i=1}^n r_i + \alpha_{\hat{C}} \mid r \geq \sum_{i=1}^n r_i + \alpha_{\hat{C}}\} \leq r
\end{aligned}$$

This inequality holds for an arbitrary value r and, consequently, for every environment ρ . So, $\llbracket - \rrbracket_{\xi}$ is an interpretation of \mathcal{H} .

Throughout the paper, we will fix the assignment of the eval_A function symbol by setting $\forall A, \xi(\text{eval}_A)(X) = X$. As illustrated by the above example, such an assignment is a reasonable choice.

Now we can show that the result of an evaluation inherits the property of the interpretation: the interpretation of an expression is an upper bound on the interpretation of its computed value.

Proposition 1. *Let \mathcal{H} and $\emptyset; \Delta \vdash e :: A$ be an environment and an expression admitting both the interpretation $\llbracket - \rrbracket_{\xi}$. Then:*

$$\mathcal{H}, e \Downarrow v \text{ implies } \llbracket e \rrbracket_{\xi} \geq \llbracket v \rrbracket_{\xi}$$

Proof. By induction on the derivation proving $\mathcal{H}, e \Downarrow v$. The base case where the derivation consists only in an application of the rule (val) is trivial. The case the derivation ends with an application of the rule (rec) follows directly by induction hypothesis.

Let us consider the case where the derivation ends with:

$$\frac{\mathcal{H}; e\{e_1/x_1, \dots, e_n/x_n\} \Downarrow v \quad (\mathbf{f} \mathbf{x}_1 \cdots \mathbf{x}_n = \mathbf{e}) \in \mathcal{H}}{\mathcal{H}; \mathbf{f}(e_1, \dots, e_n) \Downarrow v}$$

By induction hypothesis, we have $\llbracket e\{e_1/x_1, \dots, e_n/x_n\} \rrbracket_{\xi} \geq \llbracket v \rrbracket_{\xi}$ and by assumption we have $\llbracket \mathbf{f}(x_1, \dots, x_n) \rrbracket_{\xi} \geq \llbracket \mathbf{e} \rrbracket_{\xi}$. So, by several applications of Lemma 3 we obtain $\llbracket \mathbf{f}(e_1, \dots, e_n) \rrbracket_{\xi} \geq \llbracket e\{e_1/x_1, \dots, e_n/x_n\} \rrbracket_{\xi}$ and by transitivity the conclusion follows.

Let us consider the case where the derivation ends with:

$$\frac{\mathcal{H}; e \Downarrow c(e'_1, \dots, e'_m) \quad p_i = c(x_1, \dots, x_m) \quad \mathcal{H}; e_i\{e'_1/x_1, \dots, e'_m/x_m\} \Downarrow v}{\mathcal{H}; \text{Case } e \text{ of } p_1 \rightarrow e_1, \dots, p_n \rightarrow e_n \Downarrow v}$$

By induction hypothesis, we have both $\llbracket e_i\{e'_1/x_1, \dots, e'_m/x_m\} \rrbracket_{\xi} \geq \llbracket v \rrbracket_{\xi}$ and $\llbracket e \rrbracket_{\xi} \geq \llbracket c(e'_1, \dots, e'_m) \rrbracket_{\xi}$. Consider an arbitrary variable assignment ρ . Applying Lemma 1 several times, we have $\llbracket c(e'_1, \dots, e'_m) \rrbracket_{\rho, \xi} = \llbracket c(x_1, \dots, x_m) \rrbracket_{\rho', \xi}$ for $\rho' = \rho\{x_1 := \llbracket e'_1 \rrbracket_{\rho, \xi}, \dots, x_m := \llbracket e'_m \rrbracket_{\rho, \xi}\}$. By definition of assignment we have:

$$\llbracket \text{Case } e \text{ of } p_1 \rightarrow e_1, \dots, p_n \rightarrow e_n \rrbracket_{\rho, \xi} \geq \llbracket e_i \rrbracket_{\rho', \xi}$$

and by some applications of Lemma 1:

$$(\text{Case } e \text{ of } p_1 \rightarrow e_1, \dots, p_n \rightarrow e_n)_{\rho, \xi} \geq (e_i)_{\rho', \xi} = (e_i\{e'_1/x_1, \dots, e'_m/x_m\})_{\rho, \xi}$$

Since this holds for every ρ , the conclusion easily follows by the definition of interpretation. \square

The previous result can be easily extended to strict evaluation: the interpretation of an expression is an upper bound on the interpretation of its computed strict value.

Corollary 2. *Let \mathcal{H} and $\emptyset; \Delta \vdash e :: A$ be an environment and an expression admitting both the interpretation $(-)_\xi$. Then:*

$$\mathcal{H}, e \Downarrow_v \underline{v} \text{ implies } (e)_\xi \geq (\underline{v})_\xi$$

Proof. The notation $\mathcal{H}, e \Downarrow_v \underline{v}$ is just a shorthand for $\mathcal{H}, \text{eval}_A e \Downarrow \underline{v}$, so the conclusion follows directly using Proposition 1 and the fact that we consider assignments such that $\xi(\text{eval}_A)(X) = X$. \square

The last important property of interpretations that will be used in the sequel relates the size of an expression with its interpretation.

Lemma 4. *Let $(-)_\xi$ be an interpretation. Then, there exists a function $F : \mathbb{R}^+ \rightarrow \mathbb{R}^+$ such that for every program $e :: A$ admitting $(-)_\xi$:*

$$(e)_\xi \leq F(|e|)$$

Proof. By induction on the shape of e . \square

The proof of Lemma 4 proceeds in essentially the same way as the one of the subsequent Lemma 6. So, for convenience we detail only the proof of the latter.

3.3. Parametrized Interpretations

For the analyses that we will present in the next section it is convenient to extend the notion of interpretations in a parametric way. This notion allows us to obtain more precise analyses on stream programs.

The idea behind a parametrized interpretation is that the interpretations now become of the shape $(-)_\xi^l$ where $l \in \mathbb{R}$ is a parameter that can be used to refer to a particular element of a stream. In order to obtain this, we need to parametrize all the previous definitions.

Definition 14 (Parametrized Assignment).

- A parametrized symbol assignment, denoted ξ^l is a map associating to each symbol $\tau \in \mathcal{C} \cup \mathcal{F}$ and $l \in \mathbb{R}$ a function $F_l : \mathbb{R}^+ \times \dots \times \mathbb{R}^+ \rightarrow \mathbb{R}^+$ of the same arity.
- Given a variable assignment ρ , a parametrized symbol assignment ξ^l , a parametrized assignment $(-)_\xi^l$ is the extension of ρ and ξ^l to expressions defined as follows:

- $(\text{Err})_{\rho, \xi}^l = 0$
- $(\mathbf{x})_{\rho, \xi}^l = \rho(x)$
- $(\mathbf{e}_1 : \mathbf{e}_2)_{\rho, \xi}^l = \xi^l(\cdot)(\langle \mathbf{e}_1 \rangle_{\rho, \xi}^l, \langle \mathbf{e}_2 \rangle_{\rho, \xi}^{l-1})$
- $(\mathbf{t}(\mathbf{e}_1, \dots, \mathbf{e}_n))_{\rho, \xi}^l = \xi^l(\mathbf{t})(\langle \mathbf{e}_1 \rangle_{\rho, \xi}^l, \dots, \langle \mathbf{e}_n \rangle_{\rho, \xi}^l) \quad \text{for } \mathbf{t} \neq :$
- $(\text{LetRec d in } \mathbf{e})_{\rho, \xi}^l = \langle \mathbf{e} \rangle_{\rho, \xi}^l$
- $(\text{Case } \mathbf{e} \text{ of } \mathbf{c}_1(\vec{\mathbf{x}}_1) \rightarrow \mathbf{e}_1, \dots, \mathbf{c}_m(\vec{\mathbf{x}}_m) \rightarrow \mathbf{e}_m)_{\rho, \xi}^l$
 $= \max_{1 \leq i \leq m} \{ \langle \mathbf{e}_i \rangle_{\rho\{\vec{\mathbf{x}}_i = \vec{r}_i\}, \xi}^l \mid \vec{r}_i \in \mathbb{R}^+ \text{ and } \langle \mathbf{e} \rangle_{\rho, \xi}^l \geq \langle \mathbf{c}_i(\vec{\mathbf{x}}_i) \rangle_{\rho\{\vec{\mathbf{x}}_i = \vec{r}_i\}, \xi}^l \}$

The definitions given for interpretations can be easily adapted to the case of parametrized interpretations.

Definition 15 (Monotonic Parametrized Assignment). *A parametrized assignment is monotonic if for any $\mathbf{t} \in \mathcal{C} \cup \mathcal{F}$, $\xi(\mathbf{t})$ is a monotonic function, i.e. $\forall r, s \in \mathbb{R}^+$ s.t. $r \geq s$ and $\forall l, l' \in \mathbb{R}$ s.t. $l \geq l'$:*

$$\xi^l(\mathbf{t})(\dots, r, \dots) \geq \xi^{l'}(\mathbf{t})(\dots, s, \dots)$$

Definition 16 (Almost-additive and Additive Parametrized Assignment).

- *The parametrized symbol assignment ξ^l is almost-additive if $\forall \mathbf{c} \in \mathcal{C}$ of arity n but the stream constructor : we have:*

$$\xi^l(\mathbf{c})(r_1, \dots, r_n) = \sum_{i=1}^n r_i + \alpha_{\mathbf{c}}, \text{ for some constant } \alpha_{\mathbf{c}} \geq 1, \text{ whenever } n > 0.$$

$$\xi^l(\mathbf{c}) = 0, \text{ otherwise.}$$

The parametrized symbol assignment ξ^l is additive if it is almost-additive and

$$\xi^l(\cdot)(r_1, r_2) = r_1 + r_2 + \alpha, \text{ for some constant } \alpha \geq 1$$

- *A parametrized assignment $(-)_{\rho, \xi}^l$ is an almost-additive (resp. additive) assignment if the parametrized symbol assignment ξ^l is almost-additive (resp. additive).*

Differently from what happens in the case of assignments, parametrized assignments do not internalize the substitution mechanism. However, for monotonic parametrized assignment we have the following important property.

Lemma 5 (Parametrized Assignment Substitution). *Given a monotonic parametrized assignment $(-)_{\rho, \xi}^l$ and an expression $\Gamma, \mathbf{x} :: \mathbf{A}; \Delta \vdash \mathbf{e} :: \mathbf{B}$, for every expression $\Gamma; \Delta \vdash \mathbf{e}' :: \mathbf{A}$ and every $l \in \mathbb{R}$ we have:*

$$\langle \mathbf{e} \rangle_{\rho\{\mathbf{x} := \langle \mathbf{e}' \rangle_{\rho, \xi}^l\}, \xi}^l \geq \langle \mathbf{e}\{\mathbf{e}'/\mathbf{x}\} \rangle_{\rho, \xi}^l$$

Proof. By induction on the expression e . We consider just the two most interesting cases.

Consider the case $e = e_1 : e_2$. By definition we have:

$$\langle e_1 : e_2 \rangle_{\rho\{x:=\langle e' \rangle_{\rho,\xi}^l\},\xi}^l = \xi^l(\cdot)(\langle e_1 \rangle_{\rho\{x:=\langle e' \rangle_{\rho,\xi}^l\},\xi}^l, \langle e_2 \rangle_{\rho\{x:=\langle e' \rangle_{\rho,\xi}^l\},\xi}^{l-1})$$

but by monotonicity we have:

$$\begin{aligned} \xi^l(\cdot)(\langle e_1 \rangle_{\rho\{x:=\langle e' \rangle_{\rho,\xi}^l\},\xi}^l, \langle e_2 \rangle_{\rho\{x:=\langle e' \rangle_{\rho,\xi}^l\},\xi}^{l-1}) \\ \geq \xi^l(\cdot)(\langle e_1 \rangle_{\rho\{x:=\langle e' \rangle_{\rho,\xi}^l\},\xi}^l, \langle e_2 \rangle_{\rho\{x:=\langle e' \rangle_{\rho,\xi}^{l-1}\},\xi}^{l-1}) \end{aligned}$$

Since by induction hypothesis we have:

$$\xi^l(\cdot)(\langle e_1 \rangle_{\rho\{x:=\langle e' \rangle_{\rho,\xi}^l\},\xi}^l, \langle e_2 \rangle_{\rho\{x:=\langle e' \rangle_{\rho,\xi}^{l-1}\},\xi}^{l-1}) \geq \xi^l(\cdot)(\langle e_1 \{e'/x\} \rangle_{\rho,\xi}^l, \langle e_2 \{e'/x\} \rangle_{\rho,\xi}^{l-1})$$

and since by definition:

$$\xi^l(\cdot)(\langle e_1 \{e'/x\} \rangle_{\rho,\xi}^l, \langle e_2 \{e'/x\} \rangle_{\rho,\xi}^{l-1}) = \langle (e_1 : e_2) \{e'/x\} \rangle_{\rho,\xi}^l$$

the conclusion follows.

Consider now the case $e = \text{Case } e''$ of $c_1(\vec{x}_1) \rightarrow e_1, \dots, c_m(\vec{x}_m) \rightarrow e_m$. Then, by definition we have:

$$\begin{aligned} \langle e \rangle_{\rho\{x:=\langle e' \rangle_{\rho,\xi}^l\},\xi}^l &= \max_{1 \leq i \leq m} \{ \langle e_i \rangle_{\rho\{\vec{x}_i=\vec{r}_i\}\{x:=\langle e' \rangle_{\rho,\xi}^l\},\xi}^l \\ &\quad | \langle e'' \rangle_{\rho\{x:=\langle e' \rangle_{\rho,\xi}^l\},\xi}^l \geq \langle c_i(\vec{x}_i) \rangle_{\rho\{\vec{x}_i=\vec{r}_i\},\xi}^l \} \end{aligned}$$

since clearly x is disjoint from the variables in \vec{x}_i . By induction hypothesis we can show that:

$$\begin{aligned} \max_{1 \leq i \leq m} \{ \langle e_i \rangle_{\rho\{\vec{x}_i=\vec{r}_i\}\{x:=\langle e' \rangle_{\rho,\xi}^l\},\xi}^l | \langle e'' \rangle_{\rho\{x:=\langle e' \rangle_{\rho,\xi}^l\},\xi}^l \geq \langle c_i(\vec{x}_i) \rangle_{\rho\{\vec{x}_i=\vec{r}_i\},\xi}^l \} \\ \geq \max_{1 \leq i \leq m} \{ \langle e_i \{e'/x\} \rangle_{\rho\{\vec{x}_i=\vec{r}_i\},\xi}^l | \langle e'' \{e'/x\} \rangle_{\rho,\xi}^l \geq \langle c_i(\vec{x}_i) \rangle_{\rho\{\vec{x}_i=\vec{r}_i\},\xi}^l \} \end{aligned}$$

and so the conclusion follows. The other cases can be obtained similarly. \square

We are now ready to define parametrized interpretations.

Definition 17 (Parametrized Interpretation). *An expression $\Gamma; \Delta \vdash e : A$ admits a parametrized interpretation $\langle - \rangle_{\xi}^l$ if for each variable assignment ρ , the assignment $\langle - \rangle_{\rho,\xi}^l$ is monotonic and such that for each function definition $\mathfrak{f}(x_1, \dots, x_n) \doteq e'$ and for each $l \in \mathbb{R}$ the following holds:*

$$\langle \mathfrak{f}(x_1, \dots, x_n) \rangle_{\rho,\xi}^l \geq \langle e' \rangle_{\rho,\xi}^l$$

Parametrized interpretations can be extended to environment as expected and thanks to this extension it is easy to verify that parametrized interpretations behave similarly to usual interpretations with respect to program evaluation. Analogously, we will write $(\mathbf{e})_{\xi}^l \geq (\mathbf{e}')_{\xi}^l$ as a shorthand for $\forall \rho, (\mathbf{e})_{\rho, \xi}^l \geq (\mathbf{e}')_{\rho, \xi}^l$.

Similarly to the case of interpretation, we want to relate parametrized interpretations to the evaluation of programs. However, in order to do this we need to introduce a new evaluation relation counting the number of pattern matchings on stream data. Let $\mathcal{H}, \mathbf{e} \Downarrow^k \mathbf{v}$ be the relation defined in Figure 3. $\mathcal{H}, \mathbf{e} \Downarrow^k \mathbf{v}$ means that $\mathcal{H}, \mathbf{e} \Downarrow \mathbf{v}$ holds using exactly k pattern matching rules on streams for producing \mathbf{v} . We define \Downarrow_v^k in the same manner: $\mathcal{H}, \mathbf{e} \Downarrow_v^k \mathbf{v}$ if $\mathcal{H}, \text{eval}_A \mathbf{e} \Downarrow^k \mathbf{v}$.

$\frac{c \in \mathcal{C}}{\mathcal{H}; c(\mathbf{e}_1, \dots, \mathbf{e}_n) \Downarrow^0 c(\mathbf{e}_1, \dots, \mathbf{e}_n)}$	$\frac{\mathcal{H} \cup \{\mathbf{d}\}; \mathbf{M} \Downarrow^k \mathbf{v}}{\mathcal{H}; \text{LetRec } \mathbf{d} \text{ in } \mathbf{M} \Downarrow^k \mathbf{v}}$
$\frac{\mathcal{H}; \mathbf{e}\{\mathbf{e}_1/x_1, \dots, \mathbf{e}_n/x_n\} \Downarrow^k \mathbf{v} \quad (\mathbf{f} \ x_1 \ \dots \ x_n \doteq \mathbf{e}) \in \mathcal{H}}{\mathcal{H}; \mathbf{f}(\mathbf{e}_1, \dots, \mathbf{e}_n) \Downarrow^k \mathbf{v}}$	
$\frac{\mathcal{H}; \mathbf{e} \Downarrow^k c(\mathbf{e}'_1, \dots, \mathbf{e}'_m) \quad \mathbf{p}_i = c(\mathbf{x}_1, \dots, \mathbf{x}_m) \quad \mathcal{H}; \mathbf{e}_i\{\mathbf{e}'_1/x_1, \dots, \mathbf{e}'_m/x_m\} \Downarrow^{k'} \mathbf{v}}{\mathcal{H}; \text{Case } \mathbf{e} \text{ of } \mathbf{p}_1 \rightarrow \mathbf{e}_1, \dots, \mathbf{p}_n \rightarrow \mathbf{e}_n \Downarrow^{k+k'} \mathbf{v}}$	
$\frac{\mathcal{H}; \mathbf{e} \Downarrow^k \mathbf{e}'_1 : \mathbf{e}'_2 \quad \mathcal{H}; \mathbf{e}_i\{\mathbf{e}'_1/x_1, \mathbf{e}'_2/x_2\} \Downarrow^{k'} \mathbf{v}}{\mathcal{H}; \text{Case } \mathbf{e} \text{ of } \mathbf{x}_1 : \mathbf{x}_2 \rightarrow \mathbf{e}_1, \text{nil} \rightarrow \mathbf{e}_2 \Downarrow^{k+k'+1} \mathbf{v}}$	
$\frac{\mathcal{H}; \mathbf{e} \Downarrow^k c(\mathbf{e}_1, \dots, \mathbf{e}_m) \quad \forall i \leq n, \mathbf{p}_i \neq c(\mathbf{x}_1, \dots, \mathbf{x}_m)}{\mathcal{H}; \text{Case } \mathbf{e} \text{ of } \mathbf{p}_1 \rightarrow \mathbf{e}_1, \dots, \mathbf{p}_n \rightarrow \mathbf{e}_n \Downarrow^k \text{Err}}$	

Table 3: Counting stream pattern matchings

Proposition 2. *Let \mathcal{H} and $\emptyset; \Delta \vdash \mathbf{e} :: A$ be an environment and an expression admitting both the parametrized interpretation $(_)_{\xi}^l$. Then:*

$$\mathcal{H}, \mathbf{e} \Downarrow^k \mathbf{v} \text{ implies } \forall l \in \mathbb{R}, (\mathbf{e})_{\xi}^l \geq (\mathbf{v})_{\xi}^{l-k}$$

Proof. The proof is analogous to the proof of Proposition 1. The only interesting case is the one where the derivation ends with:

$$\frac{\mathcal{H}; \mathbf{e} \Downarrow^k \mathbf{e}'_1 : \mathbf{e}'_2 \quad \mathcal{H}; \mathbf{e}_i\{\mathbf{e}'_1/x_1, \mathbf{e}'_2/x_2\} \Downarrow^{k'} \mathbf{v}}{\mathcal{H}; \text{Case } \mathbf{e} \text{ of } \mathbf{x}_1 : \mathbf{x}_2 \rightarrow \mathbf{e}_1, \text{nil} \rightarrow \mathbf{e}_2 \Downarrow^{k+k'+1} \mathbf{v}}$$

By induction hypothesis, we have both $(\mathbf{e}_i\{\mathbf{e}'_1/x_1, \mathbf{e}'_2/x_2\})_{\xi}^{l-(k+1)} \geq (\mathbf{v})_{\xi}^{l-(k+1)-k'}$

and $(\mathbf{e})_{\xi}^l \geq (\mathbf{e}'_1 : \mathbf{e}'_2)_{\xi}^{l-k}$. By definition we have:

$$\begin{aligned} (\text{Case } \mathbf{e} \text{ of } \mathbf{x}_1 : \mathbf{x}_2 \rightarrow \mathbf{e}_1, \text{nil} \rightarrow \mathbf{e}_2)_{\rho, \xi}^l &\geq \max\{(\mathbf{e}_1)_{\rho\{\mathbf{x}_1=r_1, \mathbf{x}_2=r_2\}, \xi}^l \mid \\ &r_1, r_2 \in \mathbb{R}^+ \text{ and } (\mathbf{e})_{\rho, \xi}^l \geq (\mathbf{x}_1 : \mathbf{x}_2)_{\rho\{\mathbf{x}_1=r_1, \mathbf{x}_2=r_2\}, \xi}^l\} \end{aligned}$$

By induction hypothesis, we can satisfy the side condition in the max by setting $r_1 = (\mathbf{e}'_1)_{\rho, \xi}^{l-k}$ and $r_2 = (\mathbf{e}'_2)_{\rho, \xi}^{l-k-1}$ since by definition of parametrized interpretation $(\mathbf{e})_{\xi}^l \geq (\mathbf{e}'_1 : \mathbf{e}'_2)_{\xi}^{l-k} = \xi^l(\cdot)(\mathbf{e}'_1)_{\rho, \xi}^{l-k}, (\mathbf{e}'_2)_{\rho, \xi}^{l-k-1}$. So we have:

$$\begin{aligned} \max\{(\mathbf{e}_1)_{\rho\{\mathbf{x}_1=r_1, \mathbf{x}_2=r_2\}, \xi}^l \mid (\mathbf{e})_{\rho, \xi}^l \geq (\mathbf{x}_1 : \mathbf{x}_2)_{\rho\{\mathbf{x}_1=r_1, \mathbf{x}_2=r_2\}, \xi}^l\} \\ \geq (\mathbf{e}_1)_{\rho\{\mathbf{x}_1=(\mathbf{e}'_1)_{\rho, \xi}^{l-k}, \mathbf{x}_2=(\mathbf{e}'_2)_{\rho, \xi}^{l-k-1}\}, \xi}^l \end{aligned}$$

By monotonicity we have:

$$(\mathbf{e}_1)_{\rho\{\mathbf{x}_1=(\mathbf{e}'_1)_{\rho, \xi}^{l-k}, \mathbf{x}_2=(\mathbf{e}'_2)_{\rho, \xi}^{l-k-1}\}, \xi}^l \geq (\mathbf{e}_1)_{\rho\{\mathbf{x}_1=(\mathbf{e}'_1)_{\rho, \xi}^{l-k-1}, \mathbf{x}_2=(\mathbf{e}'_2)_{\rho, \xi}^{l-k-1}\}, \xi}^{l-k-1}$$

Finally, by applying Lemma 5 twice we have:

$$(\mathbf{e}_1)_{\rho\{\mathbf{x}_1=(\mathbf{e}'_1)_{\rho, \xi}^{l-k-1}, \mathbf{x}_2=(\mathbf{e}'_2)_{\rho, \xi}^{l-k-1}\}, \xi}^{l-k-1} \geq (\mathbf{e}_1\{\mathbf{e}'_1/\mathbf{x}_1, \mathbf{e}'_2/\mathbf{x}_2\})_{\rho, \xi}^{l-(k+1)}$$

By induction hypothesis we have $(\mathbf{e}_1\{\mathbf{e}'_1/\mathbf{x}_1, \mathbf{e}'_2/\mathbf{x}_2\})_{\xi}^{l-(k+1)} \geq (\mathbf{v})_{\xi}^{l-(k+1)-k'}$, so the conclusion follows.

Since this holds for every ρ and every $l \in \mathbb{R}$, the conclusion easily follows by definition of parametrized interpretation. \square

Corollary 3. *Let \mathcal{H} and $\emptyset; \Delta \vdash \mathbf{e} :: \mathbf{A}$ be an environment and an expression both admitting the parametrized interpretation $(-)_\xi^l$. Then:*

$$\mathcal{H}, \mathbf{e} \Downarrow_{\mathbf{v}}^k \underline{\mathbf{v}} \text{ implies } (\mathbf{e})_{\xi}^l \geq (\underline{\mathbf{v}})_{\xi}^{l-k}$$

Proof. Just check that we can define a parametrized interpretation of $\text{eval}_{\mathbf{A}}$ by setting $\forall l \in \mathbb{R}, \forall \mathbf{A}, \xi^l(\text{eval}_{\mathbf{A}})(X) = X$ as in Corollary 2. \square

Lemma 6. *Let $(-)_\xi^l$ be a parametrized interpretation. Then, there exists a function $G : \mathbb{R}^+ \times \mathbb{R}^+ \rightarrow \mathbb{R}^+$ such that for every program $\mathbf{e} :: \mathbf{A}$ admitting $(-)_\xi^l$ and every $l \in \mathbb{R}^+$:*

$$(\mathbf{e})_{\xi}^l \leq G(|\mathbf{e}|, l)$$

Proof. Define:

$$F(X, L) = \max_{\mathbf{t} \in \mathcal{C} \cup \mathcal{F}} (\mathbf{t})_{\xi}^L(X, \dots, X, X)$$

and $F^{n+1}(X, L) = F(F^n(X, L), L)$ and $F^0(X, L) = F(X, L)$. It can be shown by induction on the structure of \mathbf{e} that $(\mathbf{e})_{\xi}^l \leq F^{|\mathbf{e}|}(|\mathbf{e}|, l)$. If \mathbf{e} is a variable, a constructor or a function symbol of arity 0, then conclusion follows directly by definition of F , i.e $(\mathbf{e})_{\xi}^l \leq F(|\mathbf{e}|, l)$. Now, consider $\mathbf{e} = \mathbf{t} \mathbf{d}_1 \cdots \mathbf{d}_n$ and suppose $|\mathbf{d}_j| = \max_{i=1}^n |\mathbf{d}_i|$. By

induction hypothesis, $(\mathbf{d}_i)_\xi^l \leq F^{|\mathbf{d}_i|}(|\mathbf{d}_i|, l)$. There are two possibilities depending on the shape of τ . If $\tau \neq \cdot$, that is $\mathbf{e} \neq \mathbf{e}_1 : \mathbf{e}_2$, then by induction hypothesis, definition and monotonicity of F we have:

$$\begin{aligned} (\mathbf{e})_\xi^l &= \xi^l(\tau)((\mathbf{d}_1)_\xi^l, \dots, (\mathbf{d}_n)_\xi^l) \leq \xi^l(\tau)(F^{|\mathbf{d}_1|}(|\mathbf{d}_1|, l), \dots, F^{|\mathbf{d}_n|}(|\mathbf{d}_n|, l)) \\ &\leq \xi^l(\tau)(F^{|\mathbf{d}_j|}(|\mathbf{d}_j|, l), \dots, F^{|\mathbf{d}_j|}(|\mathbf{d}_j|, l)) \leq F(F^{|\mathbf{d}_j|}(|\mathbf{d}_j|, l), l) \\ &\leq F^{|\mathbf{d}_j|+1}(|\mathbf{d}_j|, l) \leq F^{|\mathbf{e}|}(|\mathbf{e}|, l) \end{aligned}$$

In the case where $\tau = \cdot$, and so $\mathbf{e} = \mathbf{e}_1 : \mathbf{e}_2$, by definition of parametrized interpretation, induction hypothesis, definition and monotonicity of F we have:

$$\begin{aligned} (\mathbf{e})_\xi^l &= (\mathbf{e}_1 : \mathbf{e}_2)_\xi^l = \xi^l(\cdot)((\mathbf{e}_1)_\xi^l, (\mathbf{e}_2)_\xi^{l-1}) \leq \xi^l(\cdot)(F^{|\mathbf{e}_1|}(|\mathbf{e}_1|, l), F^{|\mathbf{e}_2|}(|\mathbf{e}_2|, l-1)) \\ &\leq \xi^l(\cdot)(F^{|\mathbf{e}_1|}(|\mathbf{e}_1|, l), F^{|\mathbf{e}_2|}(|\mathbf{e}_2|, l)) \leq F^{|\mathbf{e}|}(|\mathbf{e}|, l) \end{aligned}$$

We let the reader checking the other cases of the induction including the technical but simple case where $\mathbf{e} = \text{Case } \mathbf{e}'$ of $c_1(\vec{x}_1) \rightarrow \mathbf{e}_1, \dots, c_m(\vec{x}_m) \rightarrow \mathbf{e}_m$. Now the conclusion follows easily by taking $G(X, L) = F^X(X, L)$. \square

4. Space Upper Bounds

4.1. Motivations

In several situations it is useful to have an estimate of the space needed to store the elements produced by a stream program. In some cases, this estimate can be obtained by considering the size of the greatest element produced as an output by the program. In other situations, unfortunately this cannot be done because there is no such a maximal element. However, an interesting estimate can be given by considering the position of the element in the stream. In functional programming, the full evaluation of a stream is never expected. A programmer will evaluate only some elements of a stream \mathbf{s} using some function like `!!` or `take`. In this case, it may be possible to derive an upper bound on the size of the elements using the output index n of the element we want to reach. For example, we know that the size of the complete evaluation of the expression $(\text{nats } \underline{0}) !! \underline{n}$, using the function symbol `nats` of Example 1, is bounded by the size of \underline{n} . Note that such a measure always exists when a stream is productive since it only consists in providing the size of the n -th output value, for each integer n .

In this section, we will show how to use interpretations to define two criteria useful to compute space estimates similar to the ones described above. The first criterion, named Local Upper Bound (LUB), will ensure that programs admitting a particular interpretation compute streams where the n -th element is bounded by a function f in n and in the size of the inputs. Thanks to Lemma 6 the criterion will provide an estimate of such an f . This criterion is named “local” because the bound relies also on the output index n . The second criterion, named Global Upper Bound (GUB), is a special case of LUB in which the output does not depend on the index n . It will ensure that programs admitting a particular interpretation compute stream elements bounded by a function f in the size of the inputs, independently of the index. Again, thanks to Lemma 4 the criterion will provide an estimate of such an f . This criterion is named

“global” because the bound holds for all the output stream elements. This situation can be illustrated by the following figure:

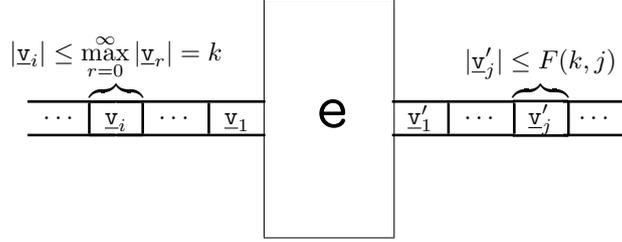


Figure 1: Local Upper Bound

A program e can be viewed as a box connecting a left tape representing the input stream and a right tape representing the output stream (we do not assume any synchrony between input and output). The program e has a *local upper bound* if each element \underline{v}'_j of the output stream has bounded size. Such a bound can depend not only on the size k of the maximal input stream element (in the case such a maximal element exists) but also on the the output element index j . In the particular case where this upper bound is independent of the index j , we say that e has a *global upper bound*. The aim of the LUB and GUB criteria we will present below is to exhibit a function $F : \mathbb{R}^+ \rightarrow \mathbb{R}^+$ witnessing these bounds.

4.2. Illustrating examples

A typical situation where programs have a local upper bound is described in the following example:

Example 5. Consider expressions built using the following definitions already presented in Example 2:

```
add :: Nat → Nat → Nat
add 0 y ≐ y
add (z + 1) y ≐ (add z y) + 1
```

```
sadd :: [Nat] → [Nat] → [Nat]
sadd (x' : xs) (y' : ys) ≐ (add x' y') : (sadd xs ys)
```

Each program will only generate output stream elements whose size is bounded by a function in their index. For example, the program:

$$e = \text{sadd } (\text{nats } \underline{3}) (\text{sadd } (\text{nats } \underline{5}) (\text{nats } \underline{4}))$$

is not globally bounded but if we evaluate the n -th output stream element as \mathcal{H} ; $e_n \Downarrow_v \underline{v}$, then we know that the size of \underline{v} is bounded by $12 + 3 \times n$. Consequently, this program has a local upper bound given by the function $F(X) = 12 + 3 \times X$ applied to the index n .

A typical situation where programs have a global upper bound is described in the following example.

Example 6. Consider expressions built using the following function definitions:

```

repeat :: Nat → [Nat]
repeat x ≐ x : (repeat x)

zip :: [a] → [a] → [a]
zip (x : xs) ys ≐ x : (zip ys xs)

square :: [Nat] → [Nat]
square (x : xs) ≐ (mul x x) : (square xs)

mul :: Nat → Nat → Nat
mul (x + 1) y ≐ add y (mul x y)
mul 0 y ≐ 0

```

Each program will only produce output stream elements whose size is bounded by some constant k . For instance, the program:

$$\text{square (zip (repeat } \underline{5} \text{) (square (zip (repeat } \underline{7} \text{) (repeat } \underline{4} \text{))))}$$

computes stream elements whose size is bounded by $k = 2401 = 7^4$. So, its global upper bound is given by the constant $k = 2401$. Now, consider the following generalization of the above program:

$$\text{square (zip (repeat } \underline{5} \text{) (square (zip x (repeat } \underline{4} \text{))))} \quad (1)$$

It is easy to verify that when x is substituted by a stream s having element sizes bounded by a constant k , then the output stream will have only elements whose sizes are bounded either by 4^4 or by k^4 . So this expression has a global upper bound given by the function F defined by $F(X) = \max(256, X^4)$.

The above examples clearly illustrate that a global upper bound implies a local upper bound but that the converse does not hold.

4.3. Definition

More formally, we can describe the situations outlined above using the formal framework presented in Section 2 as follows.

Definition 18 (Local and Global Upper Bounds). A stream program $e :: [\sigma]$ has a local upper bound if there is a function $F \in \mathbb{R}^+ \rightarrow \mathbb{R}^+$ such that:

$$\forall \underline{n} \in \mathbb{N}, \text{ if } \mathcal{H}; e_{\underline{n}} \Downarrow_v \underline{v} \text{ then } F(|\underline{n}|) \geq |\underline{v}|$$

In the case the function F is constant, then $e :: [\sigma]$ has also a global upper bound.

Note that the above definition can be used to analyze a stream program e containing both stream functions and stream definitions (programs with stream functions are provided in Examples 11 and 12). Now consider the following examples in order to illustrate this definition.

Example 7. Consider again the stream definition of `nats`:

```
nats :: Nat → [Nat]
nats x ≐ x : (nats (x + 1))
```

Clearly, `nats e` produces a stream whose elements are of unbounded size. However it is easy to verify that $\forall \underline{n} \in \mathbb{N}$, if $\mathcal{H}; (\text{nats } e)_{\underline{n}} \Downarrow v$ then $v = \underbrace{((e+1) + \dots)}_{n \text{ times}} + 1$.

Consequently, by taking $F(X) = 2 \times X$, the following inequalities are satisfied $\forall \underline{n} \in \mathbb{N}$:

$$|v| = |\underline{n}| + |e| \leq 2 \times \max(|\underline{n}|, |e|) = F(\max(|\underline{n}|, |e|))$$

Example 8. Consider the program `ones` defined as:

```
ones ≐ 1 : ones
```

Clearly, it has an obvious global upper bound ($K = 1$). Analogously, consider the program `repeat n` for every $\underline{n} \in \mathbb{N}$ where:

```
repeat x ≐ x : (repeat x)
```

Clearly, `repeat n` has a global upper bound that can be given by the function $F(X) = X$. That is, for every \underline{n} we have a constant $K_{\underline{n}} = F(|\underline{n}|) = |\underline{n}|$. In the same spirit, going back to Equation 1 of Example 6, the function $F(X) = \max(256, X^4)$ provides for every input stream `s` of data of size bounded by k a constant $K_s = \max(256, k^4)$.

4.4. LUB and GUB criteria

To ensure a Local Upper Bound we present a combined criterion consisting in a semantic condition on programs and in a semantic condition on interpretations.

Concerning the criterion on programs, we need to identify a restricted class of programs that we dub *linear programs*. These are programs that produce outputs with only a linear number of reads (stream pattern matchings in our concern). We can define them formally as follows.

Definition 19 (Linear program). Let \downarrow^k and \downarrow_v^k be the relations defined by $\mathcal{H}; e \downarrow^k v$ if there exists $k' \leq k$ such that $\mathcal{H}; e \downarrow^{k'} v$ and $\mathcal{H}; e \downarrow_v^k v$ if there exists $k' \leq k$ such that $\mathcal{H}; e \downarrow_v^{k'} v$, respectively.

A program $e :: [\sigma]$ is linear if there is a $k \geq 1$ such that for all $\underline{n} \in \mathbb{N}$, $\mathcal{H}; e_{\underline{n}} \downarrow_v^{k \times (|\underline{n}|+1)} \underline{v}$ holds. The constant k is called the linearity constant.

Now we are ready to define our criterion.

Definition 20 (LUB Criterion). A program $e :: [\sigma]$ is LUB if it is linear and it admits a parametrized interpretation $(-)_\xi^l$ that is almost-additive and such that:

$$\xi^l(\cdot)(X, Y) = \max(X, Y)$$

Now we can provide a similar criterion for Global Upper Bound:

Definition 21 (GUB Criterion). A program $e :: [\sigma]$ is GUB if it admits an interpretation $(-)_\xi$ that is almost-additive and such that:

$$\xi(\cdot)(X, Y) = \max(X, Y)$$

4.5. Soundness

Now we want to show that if a given program $e :: [\sigma]$ is LUB (resp. GUB) then it has a local (resp. global) upper bound. For that purpose, we first show an intermediate technical lemma.

Lemma 7. *Given an expression $e :: [\sigma]$:*

1. *If e is a GUB program then $\forall \underline{n} \in \mathbb{N}$, s.t. $\mathcal{H}; e_{\underline{n}} \Downarrow_v \underline{v}$ we have:*

$$\langle e \rangle_{\xi} \geq \langle \underline{v} \rangle_{\xi}$$

2. *If e is a LUB program of linearity constant k then $\forall \underline{n} \in \mathbb{N}$, s.t. $\mathcal{H}; e_{\underline{n}} \Downarrow_v \underline{v}$ we have:*

$$\langle e \rangle_{\xi}^{k \times (\underline{n} + 1)} \geq \langle \underline{v} \rangle_{\xi}^0$$

Proof. (1) We proceed by induction on $\underline{n} \in \mathbb{N}$.

Let $\underline{n} = \underline{0}$ and $\mathcal{H}; e_{\underline{0}} \Downarrow_v \underline{v}$. Then, necessarily we have a value v' such that $\mathcal{H}; e \Downarrow v'$. We have three cases, either $v' = \text{Err}$ or $v' = \text{nil}$ or $v' = e' : e''$. The former two cases are trivial. For the latter, by definition of GUB and by Proposition 1 we have:

$$\langle e \rangle_{\xi} \geq \langle e' : e'' \rangle_{\xi} = \langle \cdot \rangle_{\xi}(\langle e' \rangle_{\xi}, \langle e'' \rangle_{\xi}) \geq \langle e' \rangle_{\xi}$$

Since we clearly have $\mathcal{H}; e' \Downarrow_v \underline{v}$, by applying Corollary 1 we obtain $\langle e' \rangle_{\xi} \geq \langle \underline{v} \rangle_{\xi}$. So by transitivity we can conclude $\langle e \rangle_{\xi} \geq \langle \underline{v} \rangle_{\xi}$.

Now, let $\underline{n} = \underline{n}' + 1$ and $\mathcal{H}; e_{(\underline{n}'+1)} \Downarrow_v \underline{v}$. Again we have a value v' such that $\mathcal{H}; e \Downarrow v'$. The case $v' = \text{Err}$ is trivial. So, consider the case $v' = e' : e''$. Again by definition of GUB and by Proposition 1 we have:

$$\langle e \rangle_{\xi} \geq \langle e' : e'' \rangle_{\xi} = \langle \cdot \rangle_{\xi}(\langle e' \rangle_{\xi}, \langle e'' \rangle_{\xi}) \geq \langle e'' \rangle_{\xi}$$

Moreover $\mathcal{H}; e_{(\underline{n}'+1)} \Downarrow_v \underline{v}$ implies by definition that $\mathcal{H}; e''_{\underline{n}'} \Downarrow_v \underline{v}$ and by induction hypothesis we have $\langle e'' \rangle_{\xi} \geq \langle \underline{v} \rangle_{\xi}$. So we can conclude $\langle e \rangle_{\xi} \geq \langle \underline{v} \rangle_{\xi}$.

(2) Assume that $e :: [\sigma]$ is a LUB program of linearity constant k . We proceed by induction on $\underline{n} \in \mathbb{N}$.

Consider the base case where $\underline{n} = \underline{0}$. By assumption, we have $\mathcal{H}; e_{\underline{0}} \Downarrow_v \underline{v}$ and, necessarily we have a value v' such that $\mathcal{H}; e \Downarrow v'$. We have three cases, either $v' = \text{Err}$ or $v' = \text{nil}$ or $v' = e' : e''$. The former two cases are trivial. For the latter, by definition of linear program with linearity constant k , we know that $\mathcal{H}; e \Downarrow^{k'} e' : e''$, for some e' such that $\mathcal{H}; e' \Downarrow_v^{k''} \underline{v}$ with $k' + k'' \leq k$. Consequently, by Proposition 2 we have:

$$\langle e \rangle_{\xi}^{k \times (|\underline{0}| + 1)} = \langle e \rangle_{\xi}^k \geq \langle e' : e'' \rangle_{\xi}^{k - k'}$$

By definition it is easy to verify that:

$$\langle e' : e'' \rangle_{\xi}^{k - k'} \geq \langle e' \rangle_{\xi}^{k - k'}$$

and by Corollary 3 and monotonicity (since $k' + k'' \leq k$) we obtain:

$$\langle e' \rangle_{\xi}^{k - k'} \geq \langle \underline{v} \rangle_{\xi}^{k - k' - k''} \geq \langle \underline{v} \rangle_{\xi}^0$$

Now we prove the induction step for $\underline{n}' + 1 = \underline{n}$. Suppose that $\mathcal{H}; \mathbf{e}_{\underline{n}} \Downarrow_v^{k \times (\underline{n}+1)} \underline{v}$. It is easy to verify that necessarily $\mathcal{H}; \mathbf{e} \Downarrow^j \mathbf{e}' : \mathbf{e}''$ and $\mathcal{H}; \mathbf{e}_{\underline{n}'} \Downarrow_v \underline{v}$ for some $j, j < k$. By applying Proposition 2 we have:

$$\langle \mathbf{e} \rangle_{\xi}^{(\underline{n}+1) \times k} \geq \langle \mathbf{e}' : \mathbf{e}'' \rangle_{\xi}^{(\underline{n}+1) \times k - j}$$

By definition it is easy to verify that:

$$\langle \mathbf{e}' : \mathbf{e}'' \rangle_{\xi}^{(\underline{n}+1) \times k - j} \geq \langle \mathbf{e}'' \rangle_{\xi}^{(\underline{n}+1) \times k - (j+1)}$$

and since $j + 1 \leq k$, by monotonicity:

$$\langle \mathbf{e}'' \rangle_{\xi}^{(\underline{n}+1) \times k - (j+1)} \geq \langle \mathbf{e}'' \rangle_{\xi}^{k \times (\underline{n}' + 1)}$$

Now, applying induction hypothesis we have:

$$\langle \mathbf{e}'' \rangle_{\xi}^{k \times (\underline{n}' + 1)} \geq \langle \underline{v} \rangle_{\xi}^0$$

and so the conclusion follows. \square

We can now prove the main result of this section.

Theorem 1. *If a program is LUB (GUB) then it admits a local (global) upper bound.*

Proof. Consider a LUB program $\mathbf{e} :: [\sigma]$ wrt the parametrized interpretation $\langle - \rangle_{\xi}^l$. By Lemma 6, there is a function $G : \mathbb{R}^+ \times \mathbb{R}^+ \rightarrow \mathbb{R}^+$ such that $\forall l \in \mathbb{R}^+, G(|\mathbf{e}|, l) \geq \langle \mathbf{e} \rangle_{\xi}^l$. Let us take $F(X) = G(|\mathbf{e}|, k \times (X+1))$, k being the linearity constant of \mathbf{e} , and assume for $\underline{n} \in \mathbb{N}$ that $\mathbf{e}_{\underline{n}} \Downarrow_v \underline{v}$. By Lemma 7(2), we have $\langle \mathbf{e} \rangle_{\xi}^{k \times (\underline{n}+1)} \geq \langle \underline{v} \rangle_{\xi}^0$. Moreover, since $\langle - \rangle_{\xi}^0$ has a fixed parameter, it corresponds to an almost-additive interpretation. So, by Corollary 1 we have $\langle \underline{v} \rangle_{\xi}^0 \geq |\underline{v}|$. Summing up, we have:

$$F(|\underline{n}|) = G(|\mathbf{e}|, k \times (|\underline{n}| + 1)) \geq \langle \mathbf{e} \rangle_{\xi}^{k \times (|\underline{n}| + 1)} \geq \langle \underline{v} \rangle_{\xi}^0 \geq |\underline{v}|$$

and so the conclusion follows.

Now consider a GUB program $\mathbf{e} :: [\sigma]$. By Lemma 4 there is a function $F : \mathbb{R}^+ \rightarrow \mathbb{R}^+$ such that $F(|\mathbf{e}|) \geq \langle \mathbf{e} \rangle_{\xi}$. Let us take $K = F(|\mathbf{e}|)$ and assume for $\underline{n} \in \mathbb{N}$ that $\mathcal{H}; \mathbf{e}_{\underline{n}} \Downarrow_v \underline{v}$. By Lemma 7(1) we have $\langle \mathbf{e} \rangle_{\xi} \geq \langle \underline{v} \rangle_{\xi}$. Moreover, by Corollary 1 we have $\langle \underline{v} \rangle_{\xi} \geq |\underline{v}|$, since such that ξ is an almost-additive symbol assignment. So, summing up, we have:

$$K = F(|\mathbf{e}|) \geq \langle \mathbf{e} \rangle_{\xi} \geq \langle \underline{v} \rangle_{\xi} \geq |\underline{v}|$$

and the conclusion follows. \square

Thanks to the above theorem, if we can find a LUB interpretation for a program \mathbf{e} , then we also have a local upper bound. Let us consider some examples.

Example 9. Consider again the stream definition of `nats` of Example 1:

```
nats :: Nat → [Nat]
nats x ≐ x : (nats (x + 1))
```

We want to show that `nats` is LUB. First, notice that the program `nats` is linear with linearity constant $k = 1$. Indeed, the definition of `nats` does not involve pattern matching on stream data so the only pattern matchings correspond to the `!!` definition where one read is needed to produce on output. Now, consider the parametrized interpretation $\llbracket - \rrbracket_{\rho, \xi}^l$ defined by: $\xi^l(\text{nats})(X) = X + l$, $\xi^l(+1)(X) = X + 1$, $\xi^l(0) = 0$ and $\xi^l(:)(X, Y) = \max(X, Y)$. We check that $\forall l \in \mathbb{R}$:

$$\begin{aligned} \llbracket \text{nats } x \rrbracket_{\rho, \xi}^l &= \llbracket \text{nats} \rrbracket_{\rho, \xi}^l(\llbracket x \rrbracket_{\rho, \xi}^l) = \rho(x) + l \\ &\geq \max(\rho(x), (\rho(x) + 1) + (l - 1)) \\ &= \max(\llbracket x \rrbracket_{\rho, \xi}^l, \llbracket \text{nats}(x + 1) \rrbracket_{\rho, \xi}^{l-1}) = \llbracket x : \text{nats}(x + 1) \rrbracket_{\rho, \xi}^l \end{aligned}$$

The interpretation $\llbracket - \rrbracket_{\rho, \xi}^l$ clearly respects the required criterion for `nats` to be LUB.

That is, it is almost-additive and it is defined on $:$ as $\xi^l(:)(X, Y) = \max(X, Y)$.

So, `nats` admits a local upper bound. We obtain the required bound by setting $F(X) = \llbracket \text{nats}(\underline{m}) \rrbracket_{\rho, \xi}^X = X + \llbracket \underline{m} \rrbracket_{\rho, \xi} = X + |\underline{m}|$, for all canonical numerals \underline{m} , $\underline{n} \in \mathbb{N}$ such that $(\text{nats } \underline{m}) \text{ !! } \underline{n} \Downarrow_v \underline{v}_n$, the following holds $F(|\underline{n}|) \geq |\underline{n}| + |\underline{m}| \geq |\underline{v}_n|$ (Indeed for all \underline{n} , $\underline{v}_n = \underline{m} + \underline{n}$).

Example 10 (Fibonacci). The following example computes the Fibonacci sequence:

```
tail :: [a] → a
tail x : xs ≐ xs

fib :: [Nat]
fib ≐ 0 : (1 : (sadd fib (tail fib)))
```

We want to show that this program is LUB. First, notice that `fib` is linear with linearity constant $k = 4$. Now, consider the parametrized interpretation $\llbracket - \rrbracket_{\rho, \xi}^l$ defined by: $\xi^l(0) = 0$, $\xi^l(+1)(X) = X + 1$, $\xi^l(:)(X, Y) = \max(X, Y)$, $\xi^l(\text{sadd})(X, Y) = \xi^l(\text{add})(X, Y) = X + Y$, $\xi^l(\text{tail})(X) = X$ and $\xi^l(\text{fib}) = 2^l$. For the first rule and for each $l \in \mathbb{R}$, the following inequalities are satisfied:

$$\begin{aligned} \llbracket \text{fib} \rrbracket_{\xi}^l = 2^l &\geq \max(0, 1, 2 \times 2^{l-2}) \\ &= \max(\llbracket 0 \rrbracket_{\xi}^l, \max(\llbracket 1 \rrbracket_{\xi}^{l-1}, 2 \times \llbracket \text{fib} \rrbracket_{\xi}^{l-2})) \\ &= \max(\llbracket 0 \rrbracket_{\xi}^l, \max(\llbracket 1 \rrbracket_{\xi}^{l-1}, \llbracket \text{sadd fib (tail fib)} \rrbracket_{\xi}^{l-2})) \\ &= \max(\llbracket 0 \rrbracket_{\xi}^l, \llbracket 1 : \text{sadd fib (tail fib)} \rrbracket_{\xi}^{l-1}) \\ &= \llbracket 0 : (1 : \text{sadd fib (tail fib)}) \rrbracket_{\xi}^l \end{aligned}$$

We let the reader check the inequalities for the other definitions. So, we have that $\llbracket - \rrbracket_{\rho, \xi}^l$ respects the required criterion for `fib` to be LUB. That is, it is almost-additive and it is defined on $:$ as $\xi^l(:)(X, Y) = \max(X, Y)$. Consequently, `fib` admits a local upper

bound. The function 2^l is a parametrized upper bound on the Fibonacci sequence: for each canonical numeral $\underline{n} \in \mathbb{N}$ s.t. $\text{fib} !! \underline{n} \Downarrow_v \underline{v}_n$, the inequality $2^{4 \times (|\underline{n}|+1)} \geq |\underline{v}_n|$ is satisfied.

In the same way, if we can find a GUB interpretation for a program e , then we also have a global upper bound. Let us consider some examples.

Example 11 (Thue-Morse sequence). *The following morse program computes the Thue-Morse sequence:*

```

morse :: [Nat]
morse ≐ 0 : (zip (inv morse) (tail morse))

tail :: [a]
tail x : xs ≐ xs

inv :: [Nat] → [Nat]
inv 0 : xs ≐ 1 : xs
inv 1 : xs ≐ 0 : xs

```

The morse program is GUB with respect to the following interpretation: $\llbracket 0 \rrbracket_\xi = 0$, $\llbracket +1 \rrbracket_\xi(X) = 1 + X$, $\llbracket \cdot \rrbracket_\xi(X, Y) = \llbracket \text{zip} \rrbracket_\xi = \max(X, Y)$, $\llbracket \text{inv} \rrbracket_\xi(X) = \max(1, X)$, $\llbracket \text{tail} \rrbracket_\xi(X) = X$ and $\llbracket \text{morse} \rrbracket_\xi = 1$. For instance, for the first rule the following inequality is satisfied:

$$\begin{aligned}
\llbracket \text{morse} \rrbracket_\xi = 1 &\geq \max(0, 1, 1, 1) \\
&= \max(\llbracket 0 \rrbracket_\xi, \max(1, \llbracket \text{morse} \rrbracket_\xi, \llbracket \text{morse} \rrbracket_\xi)) \\
&= \max(\llbracket 0 \rrbracket_\xi, \max(\llbracket \text{inv morse} \rrbracket_\xi, \llbracket \text{tail morse} \rrbracket_\xi)) \\
&= \max(\llbracket 0 \rrbracket_\xi, \llbracket (\text{zip (inv morse) (tail morse)}) \rrbracket_\xi) \\
&= \llbracket 0 : (\text{zip (inv morse) (tail morse)}) \rrbracket_\xi
\end{aligned}$$

We let the reader check the inequalities for the other definitions.

Note that the Thue-Morse program can be easily checked to have a global upper bound by looking to the (finite) output range even without using the criterion. However, this example shows that the analysis can be done in a modular way. Moreover it shows that even if only simple functions are used in interpretations some interesting examples can be captured.

Example 12. *The program of Example 6 is GUB with respect to the interpretation $\llbracket - \rrbracket_{\rho, \xi}$ defined by:*

$$\begin{aligned}
\llbracket 0 \rrbracket_\xi &= 0 \\
\llbracket +1 \rrbracket_\xi(X) &= X + 1 \\
\llbracket \text{add} \rrbracket_\xi(X, Y) &= X + Y \\
\llbracket \text{zip} \rrbracket_\xi(X, Y) &= \xi(\cdot)(X, Y) = \max(X, Y) \\
\llbracket \text{square} \rrbracket_\xi(X) &= X^2 \\
\llbracket \text{mul} \rrbracket_\xi(X, Y) &= X \times Y \\
\llbracket \text{repeat} \rrbracket_\xi(X) &= X
\end{aligned}$$

Indeed, we can check the inequalities of the interpretation is satisfied for every definition and for every variable assignment ρ . For the definition of `repeat` we have:

$$\begin{aligned}
\llbracket \text{repeat } \mathbf{x} \rrbracket_{\rho, \xi} &= \llbracket \text{repeat} \rrbracket_{\xi} (\llbracket \mathbf{x} \rrbracket_{\rho, \xi}) \\
&= \rho(\mathbf{x}) \\
&\geq \max(\rho(\mathbf{x}), \rho(\mathbf{x})) \\
&= \xi(\cdot) (\llbracket \mathbf{x} \rrbracket_{\rho, \xi}, \llbracket \text{repeat } \mathbf{x} \rrbracket_{\rho, \xi}) \\
&= \llbracket \mathbf{x} : \text{repeat } \mathbf{x} \rrbracket_{\rho, \xi}
\end{aligned}$$

For the definition of `zip`:

$$\text{zip } \mathbf{z} \ \mathbf{y}\mathbf{s} \doteq \text{Case } \mathbf{z} \ \text{of } (\mathbf{x} : \mathbf{x}\mathbf{s}) \rightarrow \mathbf{x} : (\text{zip } \mathbf{y}\mathbf{s} \ \mathbf{x}\mathbf{s})$$

we check the following inequality:

$$\begin{aligned}
\llbracket \text{zip } \mathbf{z} \ \mathbf{y}\mathbf{s} \rrbracket_{\rho, \xi} &= \max(\rho(\mathbf{z}), \rho(\mathbf{y}\mathbf{s})) \\
&\geq \max\{\max(u, v, \rho(\mathbf{y}\mathbf{s})) \\
&\quad | \forall u, v \in \mathbb{R}^+ \text{ s.t. } \rho(\mathbf{z}) \geq \max(u, v)\} \\
&= \max\{\llbracket \mathbf{x} : (\text{zip } \mathbf{y}\mathbf{s} \ \mathbf{x}\mathbf{s}) \rrbracket_{\rho[x=u, \mathbf{x}\mathbf{s}=v], \xi} \\
&\quad | \forall u, v \in \mathbb{R}^+ \text{ s.t. } \llbracket \mathbf{z} \rrbracket_{\xi} \geq \llbracket \mathbf{x} : \mathbf{x}\mathbf{s} \rrbracket_{\rho[x=u, \mathbf{x}\mathbf{s}=v], \xi}\} \\
&= \llbracket \text{Case } \mathbf{z} \ \text{of } (\mathbf{x} : \mathbf{x}\mathbf{s}) \rightarrow \mathbf{x} : (\text{zip } \mathbf{y}\mathbf{s} \ \mathbf{x}\mathbf{s}) \rrbracket_{\rho, \xi}
\end{aligned}$$

We let the reader check that the inequalities are also satisfied for the remaining definitions. Consequently, the program of Example 6 admits a global upper bound. In this particular setting, the program:

$$\mathbf{e} = \text{square} (\text{zip} (\text{repeat } \underline{5}) (\text{square} (\text{zip} (\text{repeat } \underline{7}) (\text{repeat } \underline{4}))))$$

admits a global upper bound that is equal to:

$$\begin{aligned}
\llbracket \mathbf{e} \rrbracket_{\rho, \xi} &= \llbracket \text{zip} (\text{repeat } \underline{5}) (\text{square} (\text{zip} (\text{repeat } \underline{7}) (\text{repeat } \underline{4}))) \rrbracket_{\rho, \xi}^2 \\
&= (\max(\llbracket \text{repeat } \underline{5} \rrbracket_{\rho, \xi}, \llbracket \text{square} (\text{zip} (\text{repeat } \underline{7}) (\text{repeat } \underline{4})) \rrbracket_{\rho, \xi}))^2 \\
&= (\max(\llbracket \underline{5} \rrbracket_{\rho, \xi}, (\max(\llbracket \underline{7} \rrbracket_{\rho, \xi}, \llbracket \underline{4} \rrbracket_{\rho, \xi})^2))^2 \\
&= \llbracket \underline{7} \rrbracket_{\rho, \xi}^4 \\
&= 7^4
\end{aligned}$$

and we obtain that for all $\underline{n} \in \mathbb{N}$ such that $\mathcal{H}, \mathbf{e}_{\underline{n}} \Downarrow_v \underline{v}_{\underline{n}}, 7^4 \geq |\underline{v}_{\underline{n}}|$.

5. Bounded Input/Output Properties

5.1. Motivations

In this section, we show how interpretations can be used to ensure stream properties relating input reads to output writes. In particular, we are interested in estimating the

ability of a program to return a certain finite amount of elements in the output stream when fed with some (finite part of the) input stream. Giving an upper bound on the quantities of information needed can be particularly useful to implement the program in an efficient way with respect to the needed memory.

Since we are interested in the dependencies with respect to the inputs, the properties we will analyze in this section mainly concern stream functions (whereas the properties presented in the previous sections also concern stream definitions). We will concentrate on two properties of stream functions:

- The length based I/O upper bound that provides an upper bound on the number of written output stream elements in the number of read input elements.
- The size based I/O upper bound, a more precise and general notion, that provides an upper bound on the number of written output stream elements in both the number and the size of read input elements.

The size based I/O upper bound is illustrated in Figure 2.

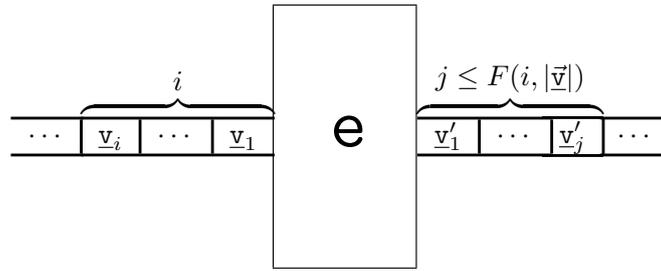


Figure 2: Size based Input/Output upper bound

Here a stream function e after reading i elements from the input produces j elements of the output. What we want is to obtain a relation linking j to i . In particular we want a function F describing an upper bound $F(i, |\vec{v}|)$ on j with respect to i and $|\vec{v}|$. We dub this kind of bound *size based* because the upper bound may depend also on the size of the input elements. In the particular case, where the function F is independent from $|\vec{v}|$, we obtain a length based upper bound. We dub this kind of bound *length based* because it only relies on the length i of the input (i.e. the number of input reads).

These properties are of finite nature. For this reason, we will define them in terms of finite stream fragments (i.e. finite lists) in a way that is reminiscent of Bird and Wadler's *Take Lemma* [5].

Notation for contexts. For notational convenience, let $e(x)$ be a notation for the expression e where the free variable x is explicitly mentioned. Let $e(\underline{v})$ be a notation for $e(x)\{\underline{v}/x\}$ and, finally, define $\llbracket e \rrbracket_{\rho, \xi}(r)$ by $\llbracket e \rrbracket_{\rho, \xi}(r) = \llbracket e(x) \rrbracket_{\rho\{x=r\}, \xi}$.

5.2. Illustrating examples

A typical situation where programs have a length based Input/Output upper bound is described in the following example:

Example 13. Consider stream expressions defined in terms of the following definitions:

$$\begin{aligned} \text{merge} &:: [a] \rightarrow [a] \rightarrow [a \times a] \\ \text{merge } (x : xs) (y : ys) &\doteq (x, y) : (\text{merge } ys xs) \\ \text{dup} &:: [a] \rightarrow [a] \\ \text{dup } (x : xs) &\doteq x : (x : (\text{dup } xs)) \end{aligned}$$

It is easy to verify that each such an expression will only generate a number of output elements related to the number of input read elements. For example, the expression:

$$\text{dup } (\text{merge } (\text{dup } s) (\text{dup } s))$$

for each read element of the input stream s computes a number of output elements bounded by $k = 4$. Consequently, $F(i) = 4 \times i$ in this particular case.

A situation where programs have a size based Input/Output upper bound is described in the following example:

Example 14. Consider stream expressions defined in terms of the following definitions:

$$\begin{aligned} \text{app} &:: [a] \rightarrow [a] \rightarrow [a] \\ \text{app } (x : xs) ys &\doteq x : (\text{app } xs ys) \\ \text{app nil } ys &\doteq ys \\ \text{upto} &:: \text{Nat} \rightarrow [\text{Nat}] \\ \text{upto } 0 &\doteq \text{nil} \\ \text{upto } (x + 1) &\doteq (x + 1) : (\text{upto } x) \\ \text{extendupto} &:: [\text{Nat}] \rightarrow [\text{Nat}] \\ \text{extendupto } (x : xs) &\doteq \text{app } (\text{upto } x) (\text{extendupto } xs) \end{aligned}$$

It is easy to verify that such expressions will only generate a number of output elements related to the number and the size of input read elements. For example, the expression:

$$\text{extendupto } (\text{extendupto } s)$$

it performs $\sum_{i=1}^{|\underline{n}|} i$ output writes for each number \underline{n} it reads on the input stream s . Consequently, it has no length based Input/Output upper bound.

5.3. Definition

More formally, we can describe the situations outlined above as follows:

Definition 22 (Length and Size Based I/O Upper Bounds).

- A stream function $x :: [\sigma] \vdash e(x) :: [\tau]$ has a length based I/O upper bound if there is a function $F : \mathbb{R}^+ \rightarrow \mathbb{R}^+$ such that for every expression $s :: [\sigma]$ we have:

$$\forall \underline{n} \in \mathbb{N}, \text{ if } \mathcal{H}; s \upharpoonright_{\underline{n}} \Downarrow_v \underline{v} \text{ and } \mathcal{H}; \text{lg } e(\underline{v}) \Downarrow_v \underline{m} \text{ then } F(|\underline{n}|) \geq |\underline{m}|$$

- A stream function $\mathbf{x} :: [\sigma] \vdash \mathbf{e}(\mathbf{x}) :: [\tau]$ has a size based I/O upper bound if there is a function $F : \mathbb{R}^+ \rightarrow \mathbb{R}^+$ such that for every expression $\mathbf{s} :: [\sigma]$ we have:

$$\forall \underline{n} \in \mathbb{N}, \text{ if } \mathcal{H}; \mathbf{s} \upharpoonright_{\underline{n}} \Downarrow_v \underline{v} \text{ and } \mathcal{H}; \mathbf{1g} \mathbf{e}(\underline{v}) \Downarrow_v \underline{m} \text{ then } F(|\underline{v}|) \geq |\underline{m}|$$

Note that for simplicity we have defined the length and size based I/O upper bounds only in the case of a unary function. However, the above definition can be easily extended to the case of functions with multiple arguments. Notice that the size based I/O property informally generalizes the length based one, i.e. a size based I/O upper bounded program is also length based I/O program, just because size always bounds the length. However, in some situations it is preferable to have the uniformity given by the length based I/O upper bound.

5.4. LBUB and SBUB criteria

We now want to introduce two criteria ensuring that a stream function has a length and size based I/O upper bounds. The definitions of length and size based I/O upper bounds above have been given in terms of the $\mathbf{1g}$ and \mathbf{take} function symbols. So, for simplicity in what follows we suppose that the considered stream functions do not use neither $\mathbf{1g}$ symbol nor \mathbf{take} symbol.

Definition 23 (LBUB and SBUB Interpretations).

- An interpretation $\langle _ \rangle_{\rho, \xi}$ is LBUB if $\langle _ \rangle_{\rho, \xi}$ is almost-additive with $\xi(+1)(X) = X + 1$ and such that:

$$\xi(\cdot)(X, Y) = Y + 1 \text{ (resp. } X + Y + 1)$$

- An interpretation $\langle _ \rangle_{\rho, \xi}$ is SBUB if $\langle _ \rangle_{\rho, \xi}$ is additive with $\xi(+1)(X) = X + 1$ and:

$$\xi(\cdot)(X, Y) = X + Y + 1$$

- An expression $\mathbf{x} :: [\sigma] \vdash \mathbf{e}(\mathbf{x}) :: [\tau]$ is LBUB (resp. SBUB) if it admits an interpretation $\langle _ \rangle_{\xi}$ such that $\langle _ \rangle_{\rho, \xi}$ is LBUB (resp. SBUB).

5.5. Soundness

LBUB assignments have some basic properties useful to deal with length based upper bounds. In particular, they give precise measures on natural numbers and lists as shown by the following two lemmas. Note that SBUB assignments behaves similarly on natural numbers.

Lemma 8. Given a LBUB or SBUB assignment $\langle _ \rangle_{\rho, \xi}$, for every $\underline{n} :: \mathbf{Nat}$ we have $\langle \underline{n} \rangle_{\rho, \xi} = |\underline{n}|$.

Proof. By induction on the length of \underline{n} . The base case follows easily by additivity of $\langle _ \rangle_{\rho, \xi}$. That is $|0| = \langle 0 \rangle_{\rho, \xi} = 0$. Consider now the case $\underline{n} + 1$. We have $\langle \underline{n} + 1 \rangle_{\rho, \xi} = \langle +1 \rangle_{\rho, \xi}(\langle \underline{n} \rangle_{\rho, \xi}) = \langle \underline{n} \rangle_{\rho, \xi} + 1$. By induction hypothesis $\langle \underline{n} \rangle_{\rho, \xi} = |\underline{n}|$ and so the conclusion follows. \square

Second, the interpretation of a finite list is equal to its length:

Lemma 9. *Given a LBUB assignment $\langle \!-\! \rangle_{\rho, \xi}$, for every $\underline{v} :: [\sigma]$ we have if $\mathcal{H}; \text{lg } \underline{v} \Downarrow_v \underline{m}$ then $\langle \underline{v} \rangle_{\rho, \xi} = |\underline{m}|$.*

Proof. By induction on the length of \underline{v} . The base case follows easily by almost-additivity of $\langle \!-\! \rangle_{\rho, \xi}$. That is $|0| = \langle \text{nil} \rangle_{\rho, \xi} = 0$ since $\mathcal{H}; \text{lg } \text{nil} \Downarrow_v 0$.

Consider now the case $\underline{v}_1 : \underline{v}_2$ and suppose that $\mathcal{H}; \text{lg } (\underline{v}_1 : \underline{v}_2) \Downarrow_v \underline{m} + 1$. We have $\langle \underline{v}_1 : \underline{v}_2 \rangle_{\rho, \xi} = \xi(\cdot)(\langle \underline{v}_1 \rangle_{\rho, \xi}, \langle \underline{v}_2 \rangle_{\rho, \xi}) = \langle \underline{v}_2 \rangle_{\rho, \xi} + 1$. Moreover, $\mathcal{H}; \text{lg } \underline{v}_2 \Downarrow_v \underline{m}$ and, by induction hypothesis, $\langle \underline{v}_2 \rangle_{\rho, \xi} = |\underline{m}|$ and so the conclusion follows. \square

For simplicity we have assumed that the program expressions of this section do not contain the function symbol lg . However, since our criteria are defined in terms of it, in order to prove their soundness we need to have an interpretation for it. This can be done easily. Indeed, every LBUB or SBUB assignment can be extended to accommodate an interpretation for the lg function symbol as follows.

Lemma 10. *Suppose that the environment \mathcal{H} admits the interpretation $\langle \!-\! \rangle_{\rho, \xi}$ and that $\langle \!-\! \rangle_{\rho, \xi}$ is a LBUB or SBUB assignment. Then, $\langle \!-\! \rangle_{\rho, \xi}$ can be extended to the function symbol lg by setting $\xi(\text{lg})(X) = X$ in such a way that the environment $\{\text{lg } y = \text{Case } y \text{ of } \text{nil} \rightarrow 0, \text{Err} \rightarrow 0, x : xs \rightarrow (\text{lg } xs) + 1\} \cup \mathcal{H}$ admits the interpretation $\langle \!-\! \rangle_{\rho, \xi}$.*

Proof. Consider the definition $\{\text{lg } y \doteq \text{Case } y \text{ of } \text{nil} \rightarrow 0, \text{Err} \rightarrow 0, x : xs \rightarrow (\text{lg } xs) + 1\}$. We have to check that the extension of $\langle \!-\! \rangle_{\rho, \xi}$ still satisfies the inequalities of an interpretation for this definition:

$$\begin{aligned} \langle \text{lg } y \rangle_{\rho, \xi} &= \rho(y) = r \\ &\geq \max(0, r) \\ &= \max(\max\{\langle 0 \rangle_{\rho, \xi} \mid r \geq \langle \text{nil} \rangle_{\rho, \xi}\}, \\ &\quad \max\{\langle 0 \rangle_{\rho, \xi} \mid r \geq \langle \text{Err} \rangle_{\rho, \xi}\}, \\ &\quad \max\{\langle (\text{lg } xs) + 1 \rangle_{\rho[x:=u, xs:=v], \xi} \mid \forall u, v \text{ s.t. } r \geq v + 1\}) \\ &= \langle \text{Case } y \text{ of } \text{nil} \rightarrow 0, \text{Err} \rightarrow 0, x : xs \rightarrow (\text{lg } xs) + 1 \rangle_{\rho, \xi} \end{aligned}$$

This inequality holds for every ρ such that $\rho(y) = r$, for an arbitrary $r \geq 0$, and so the conclusion. \square

We are now ready to prove that the LBUB (resp. SBUB) is a criterion to ensure the length (resp. size) based I/O upper bound.

Theorem 2. *If an expression $x :: [\sigma] \vdash e(x) :: [\tau]$ is LBUB (resp. SBUB) then it has also a length (resp. size) based I/O upper bound.*

Proof. Given a LBUB expression $e(x)$, suppose $\mathcal{H}; s \vdash_{\underline{n}} \underline{v}$ and $\mathcal{H}; \text{lg } e(\underline{v}) \Downarrow_v \underline{m}$.

By Corollary 2, we have $\langle \text{lg } e(\underline{v}) \rangle_{\rho, \xi} \geq \langle \underline{m} \rangle_{\rho, \xi}$ and, by Lemma 10, we have $\langle e(\underline{v}) \rangle_{\rho, \xi} \geq \langle \underline{m} \rangle_{\rho, \xi}$.

By Lemma 9, we have $\langle \underline{v} \rangle_{\rho, \xi} = |\underline{n}|$. Applying Lemma 8, we have $\langle \underline{m} \rangle_{\rho, \xi} = |\underline{m}|$. So, we obtain $\langle e \rangle_{\rho, \xi}(|\underline{n}|) \geq |\underline{m}|$ and by taking $F = \langle e \rangle_{\rho, \xi}$ the conclusion follows.

Now suppose that $e(x)$ is SBUB, that $\mathcal{H}; s \downarrow_n \downarrow v$ and that $\mathcal{H}; \lg e(v) \downarrow_v \underline{m}$.

By Corollary 2, we have $(\lg e(v))_\xi \geq (\underline{m})_\xi$ and by, Lemmata 8 and 10, we have $(e(v))_\xi \geq (\underline{m})_\xi = |\underline{m}|$.

By definition, we obtain $(e(v))_{\rho,\xi} = (e)_{\rho,\xi}((v)_{\rho,\xi})$. Moreover, by Lemma 2, since $(-)_\xi$ is additive we have a constant α such that $(v)_{\rho,\xi} \leq |v| \times \alpha$. So, taking $F(X) = (e)_{\rho,\xi}(X \times \alpha)$, the conclusion follows. \square

We end this section by showing that the situation presented in Example 13 can be captured using the LBUB criterion while the one in Example 14 can be captured using the SBUB criterion.

Example 15. Consider again the stream definitions presented in Example 13:

```
merge :: [a] → [a] → [a × a]
merge (x : xs) (y : ys) ≐ (x, y) : (merge ys xs)

dup :: [a] → [a]
dup (x : xs) ≐ x : (dup xs)
```

To verify that every expression built using these definitions has a length based I/O upper bound it is sufficient to verify that it admits the following LBUB interpretation:

$$\xi(\text{merge})(X, Y) = \max(X, Y) \quad \xi(\text{dup})(X) = 2X$$

As an example, for each s we have:

$$(\text{dup}(\text{merge}(\text{dup } s) (\text{dup } s)))_{\rho,\xi} = 4(|s|)_{\rho,\xi}$$

and this gives a length based I/O upper bound.

Example 16. Consider again the stream definitions of Example 14:

```
app :: [a] → [a] → [a]
app (x : xs) ys ≐ x : (app xs ys)
app nil ys ≐ ys

upto :: Nat → [Nat]
upto 0 ≐ nil
upto (x + 1) ≐ (x + 1) : (upto x)

extendupto :: [Nat] → [Nat]
extendupto (x : xs) ≐ app (upto x) (extendupto xs)
```

To show that every expression built using these definitions has a size based I/O upper bound it is enough to show that the following interpretation is SBUB:

$$\xi(\text{nil}) = (0)_\xi = 0 \quad \xi(+1)(X) = X + 1 \quad \xi(\cdot)(X, Y) = X + Y + 1$$

$$\xi(\text{app})(X, Y) = X + Y \quad \xi(\text{upto})(X) = \xi(\text{extendupto})(X) = 2 \times X^2$$

In particular, by taking $F(X) = (\text{extendupto})_\xi((\text{extendupto})_\xi(X))$ we obtain a size based upper bound for the expression:

$$\text{extendupto}(\text{extendupto } s)$$

That is $F(X) = 8 \times X^4$ is an upper bound on the number of output elements. Notice that the bound is less tight than what one would have expected. Indeed, in this example F gives a bound also on the size of produced elements.

6. Computing an Interpretation

One of the aspects of interpretation methods that makes them of practical interest is that they do not only provide techniques to *ensure the existence* of particular upper bounds but in several cases of practical interest they also actually provide a tool to effectively *compute* those upper bounds. Indeed, Lemma 4 says that if we have an interpretation of a given program M , then we can compute an upper bound on the size of the result of its evaluation. So it is natural to consider the corresponding *interpretation synthesis* problem that can be formulated as follows:

Interpretation synthesis problem: given a program M and a class of functions \mathfrak{F} , is an interpretation for M using only functions in \mathfrak{F} computable?

The ability of being able to compute an interpretation clearly depends on the class of functions \mathfrak{F} that one wants to consider. In particular as a consequence of the Rice's theorem the interpretation synthesis problem is undecidable for an unrestricted class of computable functions. More interestingly this problem becomes decidable when restricted class of functions are considered. See [34] for a survey.

Even if in this paper we do not concentrate on the synthesis problem for the different criteria we have introduced, the possibility of having efficient procedures to compute the studied bounds is a strong motivation of our work. In particular, the criteria studied here become particularly useful when the interpretations use only small classes of functions as codomain (e.g. polynomials, logarithmic or linear functions). We leave this important study for future works.

Another related problem that we have not addressed here is the complexity of the interpretation synthesis problem. This problem has been previously studied for functions coming from max-plus and max-poly algebras over integers and reals and the results obtained can be adapted to our framework (see [34]).

It is worth noting that this problem has also been studied by the rewriting community both from theoretical and practical points of view [30, 25].

7. Related works

In the data processing scenario, a greater attention is paid to the so called *streaming algorithms*. These are algorithms working with restricted computational power on huge amount of data (not necessarily infinite). Usually they have only limited access to the inputs and they have only a little amount of available memory. Moreover, they may also satisfy some timing constraints. Moreover, the criteria we have designed so far are inspired by the constraints that streaming algorithms should usually satisfy.

It is not surprising that when one wants to implement programs working on streams, one needs to pay attention to memory management. It is indeed not difficult to find examples of stream programs generating subtle buffering or overflow errors. In this

perspective, in [17] Frankau and Mycroft have proposed a framework that, starting from programs written in a first-order functional language, extracts stream program implementations avoiding unbounded buffering. In order to achieve this goal, their programs have to obey some specific linearity and stability typing disciplines. Analogously, Hughes et al. in their paper introducing *sized types* [22] show how the latter can be used to prevent errors related to memory leaks and buffer overflows of embedded programs. Even if sized types have been mainly introduced to prove termination properties of reactive systems (corresponding to productivity of stream programs) they have also found several applications in complexity analysis. For instance in [37], the authors show how to exploit sized types in a system designed to verify the resource usage of strict first-order programs working on lists. The programming language they are able to analyze through their type system is similar to the language we consider in the present paper, a key distinction being that their analysis is restricted to finite lists.

It is quite surprising that in the implicit computational complexity domain only few works have been carried so far on programs computing over infinite data structures. This is even more surprising if one thinks that usual tools of complexity theory, well behaving on finite data types, cannot be directly applied neither to streams nor to other infinite data structures. In [10] Burrell et al. have developed a sound and complete polynomial time complexity programming language, dubbed *Pola*, based on a type system with restrictions inspired by safe recursion. Interestingly, *Pola* permits the programmer to deal with polynomial time functional programs working both on inductive and coinductive data types.

In [29], Leivant and Ramyaa have proposed a framework based on equational programs and intrinsic theories, previously introduced by Leivant in [28], that is useful to reason about programs over inductive and co-inductive types. They used such a framework to obtain an implicit characterization of primitive corecurrence (a weak form of productivity). More recently, in [36] they have also shown that a ramified version of co-recurrence gives an implicit characterization of the class of functions over streams working in logarithmic space. In contrast to our approach considering functions working on data types, their characterization deals with functions working on streams of digits; however, they consider the complexity of a stream program as a function of the output. That is the space needed to compute the n -th element of the stream. With this respect, their characterization uses an approach similar to the one we follow for the Local Upper Bound property and for the Bounded Input/Output properties.

Using an approach similar to the one presented in this paper, Férée et al. [16] show that interpretations can be used on stream programs also to characterize type 2 polynomial time functions. In particular, they extend interpretations in order to use second order polynomials to characterize the set of functions computable in polynomial time by Oracle Turing Machines and by their unitary version. Thanks to this they obtain an implicit characterization of the class of the Basic Feasible Functionals of Cook and Urquhart [11].

Recently, Baillot and Dal Lago [4] have developed a technique inspired by quasi-interpretations to study the complexity of higher-order rewriting programs. In their framework infinite data are first class citizens in the form of higher order functions. However, they do not consider programs working on declarative infinite data structures as streams that are instead the focus of our work.

8. Conclusion

In this paper, we have studied some complexity properties of programs working on streams. In order to do this, in a first step we have adapted the interpretation methods to a functional programming languages able to express in a natural way stream programs. This has required to customize the definitions of interpretations, developed so far in the context of first-order term rewriting, to the more specific case of a first-order lazy functional programming language. As a byproduct, this has shown the flexibility of the interpretation tools in dealing with different object languages and in dealing with streams and infinite data types.

In a second step, we have exploited the use of interpretation methods by defining several criteria for the study of different space properties. These criteria correspond to resource static analyses useful to the programmer that would be able to control the complexity of the stream program he writes. They fall in two main categories:

- The first category, that includes local and global upper bounds, provides an upper bound on the size of each computed stream element. This upper bound can be a constant in the case of the global upper bound or a function of the output element position in the case of the local upper bound. The stream program that can be analyzed with respect to these criteria gives the guarantee that no one of its output elements will provoke a *memory overflow*.
- The second category, that includes the size and length based input and output upper bounds, provides an upper bound on the number of output elements with respect to the number or size of input reads.

The two categories above deal with two different dimensions of streams. By combining together the different criteria one can study several memory management aspects of programs working on streams.

- [1] Abramsky, S., Ong, C.-H. L., 1993. Full abstraction in the lazy lambda calculus. *Inf. Comput.* 105 (2), 159–267.
- [2] Amadio, R., 2005. Synthesis of max-plus quasi-interpretations. *Fundamenta Informaticae* 65(1–2).
- [3] Ariola, Z. M., Felleisen, M., 1997. The call-by-need lambda calculus. *J. Funct. Program.* 7 (3), 265–301.
- [4] Baillot, P., Dal Lago, U., 2012. Higher-Order Interpretations and Program Complexity. In: *Proceedings of CSL'12*. Vol. 16 of *Leibniz International Proceedings in Informatics (LIPIcs)*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 62–76.
- [5] Bird, R., Wadler, P., 1988. *Introduction to Functional Programming*. Prentice Hall.
- [6] Bonfante, G., Cichon, A., Marion, J.-Y., Touzet, H., 2001. Algorithms with polynomial interpretation termination proof. *Journal of Functional Programming* 11 (1), 33–53.

- [7] Bonfante, G., Marion, J., Moyen, J., 2011. Quasi-interpretations a way to control resources. *Theoretical Computer Science*.
- [8] Bonfante, G., Marion, J.-Y., Moyen, J.-Y., Jul 2001. On lexicographic termination ordering with space bound certifications. In: *PSI*. Vol. 2244 of LNCS. Springer.
- [9] Bonfante, G., Marion, J.-Y., Moyen, J.-Y., Péchoux, R., 2005. Synthesis of quasi-interpretations. *LCC2005, LICS Workshop*.
- [10] Burrell, M. J., Cockett, R., Redmond, B. F., August 2009. Pola: a language for PTIME programming. In: *Tenth International Workshop on Logic and Computational Complexity*. Los Angeles, USA.
- [11] Cook, S., Urquhart, A., 1989. Functional interpretations of feasibly constructive arithmetic. In: *Proceedings of the twenty-first annual ACM symposium on Theory of computing*. STOC '89. ACM, New York, NY, USA, pp. 107–112.
- [12] Coquand, T., 1994. Infinite objects in type theory. Vol. 806 of LNCS. Springer, pp. 62–78.
- [13] Cousot, P., Cousot, R., 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *Proceedings of ACM POPL'77*, 238–252.
- [14] Dijkstra, E.-W., 1980. On the productivity of recursive definitions, EWD749.
URL <http://www.cs.utexas.edu/users/EWD/ewd07xx/EWD749.PDF>
- [15] Endrullis, J., Grabmayer, C., Hendriks, D., Ishihara, A., Klop, J., 2007. Productivity of Stream Definitions. Vol. 4639 of LNCS. Springer, p. 274.
- [16] Férée, H., Hainry, E., Hoyrup, M., Péchoux, R., 2010. Interpretation of stream programs: characterizing type 2 polynomial time complexity. Springer, pp. 291–303.
- [17] Frankau, S., Mycroft, A., 2003. Stream processing hardware from functional language specifications. *Proceeding of IEEE HICSS-36*.
- [18] Gaboardi, M., Péchoux, R., 2009. Upper bounds on stream I/O using semantic interpretations. In: *CSL 2009*. Vol. 5771 of LNCS. Springer, pp. 271–286.
- [19] Gaboardi, M., Péchoux, R., 2010. Global and local space properties of stream programs. In: *1st International Workshop on Foundational and Practical Aspects of Resource Analysis – FOPARA'09*. Vol. 6324 of LNCS. pp. 51 – 66.
- [20] Gordon, A., 1999. Bisimilarity as a theory of functional programming. *TCS* 228.
- [21] Henderson, P., Morris, J., 1976. A lazy evaluator. *Proceedings of POPL'76*, 95–103.

- [22] Hughes, J., Pareto, L., Sabry, A., 1996. Proving the correctness of reactive systems using sized types. *Proceedings of ACM POPL'96*, 410–423.
- [23] Kennaway, J., Klop, J., Sleep, M., de Vries, F., 1989. Transfinite Reductions in Orthogonal Term Rewriting Systems. Vol. 488 of LNCS. pp. 1–12.
- [24] Kennaway, J., Klop, J., Sleep, M., de Vries, F., 1997. Infinitary lambda calculus. *TCS* 175 (1), 93–125.
- [25] Korp, M., Sternagel, C., Zankl, H., Middeldorp, A., 2009. Tyrolean termination tool 2. In: *RTA*. Springer, pp. 295–304.
- [26] Lankford, D., 1979. On proving term rewriting systems are noetherien, tech. rep.
- [27] Launchbury, J., 1993. A natural semantics for lazy evaluation. *Proceedings of POPL'93*, 144–154.
- [28] Leivant, D., 1995. Intrinsic theories and computational complexity. In: Leivant, D. (Ed.), *Logic and Computational Complexity*. Vol. 960 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, pp. 177–194.
- [29] Leivant, D., Ramyaa, R., 2012. Implicit complexity for coinductive data: a characterization of corecurrence. In: Marion, J.-Y. (Ed.), *Second Workshop on Developments in Implicit Computational Complexity*. Vol. 75 of *EPTCS*. Open Publishing Association, pp. 1–14.
- [30] Lucas, S., 2005. Polynomials over the reals in proofs of termination: from theory to practice. *RAIRO Theoretical Informatics and Applications* 39 (3), 547–586.
- [31] Manna, Z., Ness, S., 1970. On the termination of Markov algorithms. In: *Third hawaii international conference on system science*. pp. 789–792.
- [32] Marion, J.-Y., Péchoux, R., 2009. Sup-interpretations, a semantic method for static analysis of program resources. *ACM Transactions on Computational Logic (TOCL)* 10 (4), 27.
- [33] Ong, C.-H. L., 1992. Lazy lambda calculus: Theories, models and local structure characterization (extended abstract). In: *ICALP*. pp. 487–498.
- [34] Péchoux, R., 2013. Synthesis of sup-interpretations: A survey. *Theor. Comput. Sci.* 467, 30–52.
- [35] Pitts, A. M., 1997. Operationally-based theories of program equivalence. In: *Semantics and Logics of Computation*. Cambridge University Press.
- [36] Ramyaa, R., Leivant, D., 2011. Ramified corecurrence and logspace. *ENTCS* 276 (0), 247 – 261, *Twenty-seventh Conference on the Mathematical Foundations of Programming Semantics (MFPS XXVII)*.
- [37] Shkaravska, O., van Eekelen, M., Tamalet, A., July 2009. Collected Size Semantics for Functional Programs over Polymorphic Nested Lists. Tech. Rep. ICIS–R09003, Radboud University Nijmegen.

- [38] Sijtsma, B., 1989. On the productivity of recursive list definitions. ACM TOPLAS 11 (4), 633–649.
- [39] Stephens, R., 1997. A survey of stream processing. Acta Informatica 34 (7), 491–541.
- [40] Weihrauch, K., 1997. A foundation for computable analysis. Vol. 1337 of LNCS. Springer, pp. 185–200.

Appendix A. Comparison with quasi-interpretation

Quasi-interpretations introduced in [7] are a previous formulation of interpretations that require inequalities of the shape $\langle l \rangle_\xi \geq \langle r \rangle_\xi$ for each rule $l \rightarrow r$ of a TRS. Note that the new definition suggested in this paper is a generalization of these previous works on TRS to functional programs since the interpretation of a definition $\mathbf{f} \mathbf{x} \doteq \text{Case } \mathbf{x} \text{ of } c_1(\vec{x}_1) \rightarrow e_1, \dots, c_m(\vec{x}_m) \rightarrow e_m$ generates the following inequality:

$$\langle \mathbf{f} \mathbf{x} \rangle_{\rho, \xi} \geq \max_{i \in \{1, m\}} \{ \langle e_i \rangle_{\rho\{\vec{x}_i := \vec{r}_i\}, \xi} \mid \forall \vec{r}_i \text{ s.t. } \langle \mathbf{x} \rangle_{\rho, \xi} \geq \langle c_i(\vec{x}_i) \rangle_{\rho\{\vec{x}_i := \vec{r}_i\}, \xi} \}$$

which implies, using Lemma 3, $\forall i \in \{1, m\}$:

$$\begin{aligned} & \langle \mathbf{f} c_i(\vec{x}_i) \rangle_{\rho, \xi} \\ &= \langle \mathbf{f} \mathbf{x} \{ c_i(\vec{x}_i) / \mathbf{x} \} \rangle_{\rho, \xi} \\ &\geq \max_{i \in \{1, m\}} \{ \langle e_i \rangle_{\rho\{\vec{x}_i := \vec{r}_i\}, \xi} \mid \forall \vec{r}_i \text{ s.t. } \langle \mathbf{x} \{ c_i(\vec{x}_i) / \mathbf{x} \} \rangle_{\rho, \xi} \geq \langle c_i(\vec{x}_i) \rangle_{\rho\{\vec{x}_i := \vec{r}_i\}, \xi} \} \\ &\geq \max \{ \langle e_i \rangle_{\rho\{\vec{x}_i := \vec{r}_i\}, \xi} \mid \forall \vec{r}_i \text{ s.t. } \langle c_i(\vec{x}_i) \rangle_{\rho, \xi} \geq \langle c_i(\vec{x}_i) \rangle_{\rho\{\vec{x}_i := \vec{r}_i\}, \xi} \} \\ &\geq \langle e_i \rangle_{\rho, \xi} \end{aligned}$$

The last inequality is obtained by taking the particular values $r_i = \rho(\mathbf{x}_i)$ since, in this particular case, $\rho\{\vec{x}_i := \vec{r}_i\} = \rho\{\vec{x}_i := \rho(\mathbf{x}_i)\} = \rho$ and, consequently, the inequality $\langle c_i(\vec{x}_i) \rangle_{\rho, \xi} \geq \langle c_i(\vec{x}_i) \rangle_{\rho\{\vec{x}_i := \vec{r}_i\}, \xi}$ is satisfied. To conclude, we obtain $\forall i \in \{1, m\}$, $\langle \mathbf{f} c_i(\vec{x}_i) \rangle_\xi \geq \langle e_i \rangle_\xi$ which corresponds exactly to the notion of quasi-interpretation applied to the equivalent TRS defined by the rules $\mathbf{f} c_1(\vec{x}_1) \rightarrow e_1, \dots, \mathbf{f} c_m(\vec{x}_m) \rightarrow e_m$.

Appendix B. An example of Strict Evaluation

Consider an environment \mathcal{H} containing the functions defined in Example 1. As already stressed, the lazy evaluation produces the first bit of information. So, for instance we have:

$$\frac{\frac{\mathcal{H}; 0 : (\text{nats } (0 + 1)) \Downarrow 0 : (\text{nats } (0 + 1))}{\mathcal{H}; \text{nats } 0 \Downarrow 0 : (\text{nats } (0 + 1))} \text{ (val)}}{\text{ (fun)}}$$

When a pattern matching error occurs this is traced by the Err constant. So, since the definition of zip without syntactic sugar is:

$$\text{zip } \mathbf{x} \ \mathbf{y} \doteq \text{Case } \mathbf{x} \text{ of } \mathbf{z} : \mathbf{w} \rightarrow \mathbf{z} : (\text{zip } \mathbf{y} \ \mathbf{w})$$

then we clearly have the following evaluation:

$$\frac{\frac{\overline{\mathcal{H}; \text{nil} \Downarrow \text{nil}} \text{ (val)} \quad \text{nil} \neq \text{z} : \text{w}}{\mathcal{H}; \text{Case nil of z : w} \rightarrow \text{z} : (\text{zip nil w}) \Downarrow \text{Err}} \text{ (pm}_e\text{)}}{\mathcal{H}; \text{zip nil nil} \Downarrow \text{Err}} \text{ (fun)}$$

Note that the following evaluation does not produce a pattern matching error:

$$\frac{\frac{\overline{\mathcal{H}; 0 : \text{nil} \Downarrow 0 : \text{nil}} \quad \overline{\mathcal{H}; 0 : (\text{zip nil nil}) \Downarrow 0 : (\text{zip nil nil})}}{\mathcal{H}; \text{Case } (0 : \text{nil}) \text{ of z : w} \rightarrow \text{z} : (\text{zip nil w}) \Downarrow 0 : (\text{zip nil nil})}}{\mathcal{H}; \text{zip } (0 : \text{nil}) \text{ nil} \Downarrow 0 : (\text{zip nil nil})}$$

Now, let us consider the environment \mathcal{H}' obtained by adding to \mathcal{H} the functions definitions defining $\text{eval}_{[\text{Nat}]}$ on lists of numerals:

$$\begin{aligned} \text{eval}_{[\text{Nat}]} &:: [\text{Nat}] \rightarrow [\text{Nat}] \\ \text{eval}_{[\text{Nat}]} \quad \text{Err} &\doteq \text{Err} \\ \text{eval}_{[\text{Nat}]} \quad \text{nil} &\doteq \text{nil} \\ \text{eval}_{[\text{Nat}]} \quad (\text{x} : \text{y}) &\doteq \text{cons} (\text{eval}_{[\text{Nat}]} \text{x}) (\text{eval}_{[\text{Nat}]} \text{y}) \end{aligned}$$

where $\text{eval}_{[\text{Nat}]}$ is defined analogously using the strict function succ defined above and cons is the function symbol for the strict version of the list of natural numbers constructor:

$$\begin{aligned} \text{cons} \quad \text{Err} \quad \text{Err} &\doteq \text{Err} : \text{Err} \\ \text{cons} \quad \text{Err} \quad \text{nil} &\doteq \text{Err} : \text{nil} \\ \text{cons} \quad 0 \quad \text{nil} &\doteq 0 : \text{nil} \\ \text{cons} \quad (\text{x} + 1) \quad \text{nil} &\doteq (\text{x} + 1) : \text{nil} \\ \text{cons} \quad \text{Err} \quad (\text{y} : \text{z}) &\doteq \text{Err} : (\text{y} : \text{z}) \\ \text{cons} \quad 0 \quad (\text{y} : \text{z}) &\doteq 0 : (\text{y} : \text{z}) \\ \text{cons} \quad (\text{x} + 1) \quad (\text{y} : \text{z}) &\doteq (\text{x} + 1) : (\text{y} : \text{z}) \end{aligned}$$

By evaluating the expression $\text{eval}_{[\text{Nat}]}(\text{zip } (0 : \text{nil}) \text{ nil})$, we obtain a result distinct from the result obtained without using the $\text{eval}_{[\text{Nat}]}$ function symbol. Indeed, this allows us to explore the entire result as shown by the following derivation where some steps are omitted for clarity:

$$\frac{\frac{\frac{\frac{\vdots}{\mathcal{H}'; \text{zip nil nil} \Downarrow \text{Err}}{\vdots}}{\mathcal{H}'; \text{eval}_{[\text{Nat}]} 0 \Downarrow 0} \quad \mathcal{H}'; \text{eval}_{[\text{Nat}]}(\text{zip nil nil}) \Downarrow \text{Err}}{\vdots}}{\mathcal{H}'; \text{cons} (\text{eval}_{[\text{Nat}]} 0) (\text{eval}_{[\text{Nat}]}(\text{zip nil nil})) \Downarrow 0 : \text{Err}}}{\vdots} \quad \frac{\vdots}{\mathcal{H}'; \text{eval}_{[\text{Nat}]}(\text{zip } (0 : \text{nil}) \text{ nil}) \Downarrow 0 : \text{Err}}$$

6 A type-based complexity analysis of Object Oriented programs

A Type-Based Complexity Analysis of Object Oriented Programs

Emmanuel Hainry, Romain Péchoux

► **To cite this version:**

Emmanuel Hainry, Romain Péchoux. A Type-Based Complexity Analysis of Object Oriented Programs. Information and Computation, Elsevier, 2018, Information and Computation, 261 (1), pp.78-115. 10.1016/j.ic.2018.05.006 . hal-01712506

HAL Id: hal-01712506

<https://hal.inria.fr/hal-01712506>

Submitted on 19 Feb 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Type-Based Complexity Analysis of Object Oriented Programs

Emmanuel Hainry, Romain Pécoux

Université de Lorraine, CNRS, Inria, LORIA, Nancy, France

Abstract

A type system is introduced for a generic Object Oriented programming language in order to infer resource upper bounds. A sound and complete characterization of the set of polynomial time computable functions is obtained. As a consequence, the heap-space and the stack-space requirements of typed programs are also bounded polynomially. This type system is inspired by previous works on Implicit Computational Complexity, using tiering and non-interference techniques. The presented methodology has several advantages. First, it provides explicit big O polynomial upper bounds to the programmer, hence its use could allow the programmer to avoid memory errors. Second, type checking is decidable in polynomial time. Last, it has a good expressivity since it analyzes most object oriented features like inheritance, overload, override and recursion. Moreover it can deal with loops guarded by objects and can also be extended to statements that alter the control flow like break or return.

Keywords: Object Oriented Program, Type system, complexity, polynomial time.

1. Introduction

1.1. Motivations

In the last decade, the development of embedded systems and mobile computing has led to a renewal of interest in predicting program resource consumption. This kind of problematic is highly challenging for popular object oriented programming languages which come equipped with environments for applications running on mobile and other embedded devices (e.g. Dalvik, Java Platform Micro Edition (Java ME), Java Card and Oracle Java ME Embedded).

The current paper tackles this issue by introducing a type system for a compile-time analysis of both heap and stack space requirements of OO programs thus avoiding memory errors. This type system is also sound and complete for the set of polynomial time computable functions on the Object Oriented paradigm.

¹This work has been partially supported by ANR Project ELICA ANR-14-CE25-0005.

This type system combines ideas coming from tiering discipline, used for complexity analysis of function algebra [1, 2], together with ideas coming from non-interference, used for secure information flow analysis [3]. The current work is an extended version of [4] and is strongly inspired by the seminal paper [5].

1.2. Abstract OO language

The results of this paper will be presented in a formally oriented manner in order to highlight their theoretical soundness. For this, we will consider a generic Abstract Object Oriented language called AOO. It can be seen as a language strictly more expressive than Featherweight Java [6] enriched with features like variable updates and while loops. The language is generic enough. Consequently, the obtained results can be applied both to impure OO languages (*e.g.* Java) and to pure ones (*e.g.* SmallTalk or Ruby). Indeed, in this latter case, it just suffices to forget rules about primitive data types in the type system. Moreover, it does not depend on the implementation of the language being compiled (ObjectiveC, OCaml, Scala, ...) or interpreted (Python standard implementation, OCaml, ...). There are some restrictions: it does not handle exceptions, inner classes, generics, multiple inheritance or pointers. Hence languages such as C++ cannot be handled. However we claim that the analysis can be extended to exceptions, inner classes and generics. This is not done in the paper in order to simplify the technical analysis. The presented work captures Safe Recursion on Notation by Bellantoni and Cook [1] and we conjecture that it could be adapted to programs with higher-order functions. The intuition behind such a conjecture is just that tiers are very closely related to the ! and § modalities of light logics [7].

1.3. Intuition

The heap is represented by a directed graph where nodes are object addresses and arrows relate an object address to its field addresses. The type system splits variables in two universes: tier **0** universe and tier **1** universe. In this setting, the high security level is tier **0** while low security level is tier **1**. While tier **1** variables are pointers to nodes of the initial heap, tier **0** variables may point to newly created addresses. The information may flow from tier **1** to tier **0**, that is a tier **0** variable may depend on tier **1** variables. However the presented type system precludes flows from **0** to **1**. Indeed once a variable has stored a newly created instance, it can only be of tier **0**. Tier **1** variables are the ones that can be used either as guards of a while loop or as a recursive argument in a method call whereas tier **0** variables are just used as storages for the computed data. This is the reason why, in analogy with information-flow analysis, tier **0** is the high security level of the current setting, though this naming is opposed to the ICC standard interpretation where tier **1** is usually seen as “safer” than **0** because its use is controlled and restricted.

The polynomial upper bound is obtained as follows: if the input graph structure has size n then the number of distinct possible configurations for k tier **1** variables is at most $O(n^k)$. For this, we put some restrictions on operations that can make the memory grow : constructors for the heap and operators and method calls for the stack.

1.4. Example

Consider the following Java code duplicating the length of a boolean `BList` as an illustrating example:

```

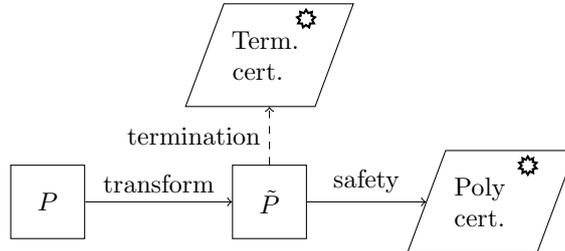
y := x.clone();
while (x != null){
    y := new BList(true,y);
    x := x.getQueue();
}

```

The tier of variable `x` will be enforced to be **1** since it is used in a while loop guard and in the call of the recursive method `clone`. On the opposite, the tier of variable `y` will be enforced to be **0** since the `y:=new BList(true,y);` instruction enlarges the memory use. For each assignment, we will check that the tier of the variable assigned to is equal to (smaller than for primitive data) the tier of the assigned expression. Consequently, the assignment `y:=x.clone();` is typable in this code (since the call `x.clone();` is of tier **0** as it makes the memory grow) whereas it cannot be typed if the first instruction is to be replaced by either `x:=y.clone();` or `x:=y`.

1.5. Methodology

The OO program complexity analysis presented in this paper can be summed up by the following figure:



In a first step, given a program P of a given OO programming language, we first apply a transformation step in order to obtain the AOO program \tilde{P} . This transformation contains the following steps:

- convert each syntactical construct of the source language in P to the corresponding construct in the abstract OO language. In particular, for statements can be replaced by while statements,
- for all public fields of P , write the corresponding getter and setter in \tilde{P} ,
- α -convert the variables so that there is no name clashes in \tilde{P} ,
- flatten the program (this will be explained in Subsection 3.5).

All these steps can be performed in polynomial time and the program abstract semantics is preserved. Consequently, P terminates iff \tilde{P} terminates.

In a second step, a termination check and a safety check can be performed in parallel. The termination certificate can be obtained using existing tools (see the related works subsection). As the semantics is preserved, the check can also be performed on the original program P or on the compiled bytecode. In the safety check, a polynomial time type inference (Proposition 5) is performed together with a safety criterion check on recursive methods. This latter check called *safety* can be checked in polynomial time. A more general criterion, called general safety and which is unfortunately undecidable, is also provided. If both checks succeed, Theorem 2 ensures polynomial time termination.

1.6. Outline

In Section 2 and 3, the syntax and semantics of the considered generic language AOO are presented. Note that the semantics is defined on meta-instructions, flattened instructions that make the semantics formal treatment easier. The main contribution: a tier based type system is presented in Section 4 together with illustrating examples. In Section 5, two criteria for recursive methods are provided: the safety that is decidable in polynomial time and the general safety that is strictly more expressive but undecidable. Section 6 establishes the main non-interference properties of the type system. Section 7 and 8 are devoted to prove soundness and, respectively, completeness of the main result: a characterization of the set of polynomial time computable functions. As an aside, explicit space upper bounds on the heap and stack space usage are also obtained. Section 9 is devoted to prove the polynomial time decidability of type inference. Section 10 discusses extensions improving the expressivity, including an extension based on declassification.

2. Syntax of AOO

In this section, the syntax of an Abstract Oriented Object programming language, called AOO, is introduced. This language is general enough so that any well-known OO programming language (Java, OCaml, Scala, ...) can be compiled to it under some slight restrictions (not all the features of these languages – threads, user interface, input/output, ... – are handled and some program transformations/refinements are needed for a practical application).

Grammar. Given four disjoint sets \mathbb{V} , \mathbb{O} , \mathbb{M} and \mathbb{C} representing the set of variables, the set of operators, the set of method names and the set of class names, respectively, expressions, instructions, constructors, methods and classes are defined by the grammar of Figure 1, where $\mathbf{x} \in \mathbb{V}$, $\mathbf{op} \in \mathbb{O}$, $\mathbf{m} \in \mathbb{M}$ and $\mathbf{c} \in \mathbb{C}$, where $[e]$ denotes some optional syntactic element e and where \bar{e} denotes a sequence of syntactic elements e_1, \dots, e_n . Also assume a fixed set of discrete primitive types \mathbb{T} to be given, *e.g.* $\mathbb{T} = \{\mathbf{void}, \mathbf{boolean}, \mathbf{int}, \mathbf{char}\}$ or $\mathbb{T} = \emptyset$ in the case of a pure OO language. In what follows, $\{\mathbf{void}, \mathbf{boolean}\} \subseteq \mathbb{T}$ will always hold.

Expressions $\ni e$	$::= x \mid \text{cst}_\tau \mid \text{null} \mid \text{this} \mid \text{op}(\bar{e})$ $\mid \text{new } C(\bar{e}) \mid e.m(\bar{e})$
Instructions $\ni I$	$::= ; \mid [\tau] x:=e; \mid I_1 I_2 \mid \text{while}(e)\{I\}$ $\mid \text{if}(e)\{I_1\}\text{else}\{I_2\} \mid e.m(\bar{e});$
Methods $\ni m_C$	$::= \tau m(\overline{\tau x})\{I[\text{return } x;]\}$
Constructors $\ni k_C$	$::= C(\overline{\tau y})\{I\}$
Classes $\ni C$	$::= C [\text{extends } D] \{\overline{\tau x}; \overline{k_C} \overline{m_C}\}$

Figure 1: Syntax of AOO classes

The τ s are type variables ranging over $\mathbb{C} \cup \mathbb{T}$. The metavariable cst_τ represents a primitive type constant of type $\tau \in \mathbb{T}$. Let $\text{dom}(\tau)$ be the domain of values of type τ . For example, $\text{dom}(\text{boolean}) = \{\text{true}, \text{false}\}$ or $\text{dom}(\text{int}) = \mathbb{N}$. $\text{cst}_\tau \in \text{dom}(\tau)$ holds. Finally, define $\text{dom}(\mathbb{T}) = \cup_{\tau \in \mathbb{T}} \text{dom}(\tau)$. Each primitive operator $\text{op} \in \mathbb{O}$ has a fixed arity n and comes equipped with a signature of the shape $\text{op} :: \tau_1 \times \dots \times \tau_n \rightarrow \tau_{n+1}$ fixed by the language implementation and such that $\tau_1, \dots, \tau_n \in \mathbb{C} \cup \mathbb{T}$ and $\tau_{n+1} \in \mathbb{T}$. That is, operator outputs are of primitive type. An example of such operator will be $=$. Also note, that primitive operators can be both considered to be applied to finite data-types as for Java integers as well as infinite datatype as for Python integers.

The AOO syntax does not include a `for` instruction based on the premise that, as in Java, a `for` statement `for(τ $x:=e$; condition; Increment){Ins}` can be simulated by the statement `τ $x:=e$; while(condition) {Ins Increment};`. Given a method $\tau m(\tau_1 x_1, \dots, \tau_n x_n)\{I [\text{return } x;]\}$ of C , its signature is $\tau m^C(\tau_1, \dots, \tau_n)$, the notation m^C denoting that m is declared in C . The signature of a constructor k_C is $C(\overline{\tau})$. Note that method overload is possible as a method name may appear in several distinct signatures.

Variables toponymy. In a class $C = C\{\tau_1 x_1; \dots; \tau_n x_n; \overline{k_C} \overline{m_C}\}$, the variables x_i are called fields. In a method or constructor $\tau m(\overline{\tau y})\{I[\text{return } x;]\}$, the arguments y_j are called parameters. Each variable x declared in an assignment of the shape $\tau x:=e$; is called a local variable. Hence, in a given class, a variable is either a field, or a parameter or a local variable. Let $C.F = \{\overline{x}\}$ to be the set of fields in a class $C \{\overline{\tau x}; \overline{k_C} \overline{m_C}\}$ and $\mathcal{F} = \cup_{C \in \mathbb{C}} C.F \subseteq \mathbb{V}$ be the set of all fields of a given program P , when P is clear from the context.

No access level. In an AOO program, the fields of an instance cannot be accessed directly using the “.” operator. Getters will be needed. This is based on the implicit assumption that all fields are `private` since there is no field access in the syntax. On the opposite, methods and classes are all `public`. This is not

a huge restriction for an OO programmer since any field can be accessed and updated in an outer class by writing the corresponding getter and setter.

Inheritance. Inheritance is allowed by the syntax of AOO programs through the use of the `extends` construct. Consequently, override is allowed by the syntax. In the case where `C extends D`, the constructors $C(\overline{\tau\overline{y}})\{I\}$ are constructors initializing both the fields of `C` and the fields of `D`. Inheritance defines a partial order on classes denoted by $C \leq D$.

AOO programs. A *program* is a collection of classes together with exactly one class `Exe`{`void main()`{`Init Comp`}} with `Init, Comp` \in Instructions. The method `main` of class `Exe` is intended to be the entry point of the program. The instruction `Init` is called the *initialization instruction*. Its purpose is to compute the program input, which is strongly needed in order to define the complexity of an AOO program (if there is no input, all terminating programs are constant time programs). The instruction `Comp` is called the *computational instruction*. The type system presented in this paper will analyze the complexity of this latter instruction. See Subsection 3.7 for more explanations about such a choice.

Well-formed programs. Throughout the paper, only well-formed programs satisfying the following conditions will be considered:

- Each class name `C` appearing in the collection of classes corresponds to exactly one class of name `C` $\in \mathbb{C}$.
- Each local variable `x` is both declared and initialized exactly once by a $\tau \ x := e;$ instruction for its first use.
- A method output type is `void` iff it has no `return` statement.
- Each method signature is unique with respect to its name, class and input types. This implies that it is forbidden to define two signatures of the shape $\tau \ m^C(\overline{\tau})$ and $\tau' \ m^C(\overline{\tau})$ with $\tau \neq \tau'$.

Example 1. *Let the class `BList` be an encoding of binary integers as lists of booleans (with the least significant bit in head). The complete code will be given in Example 8.*

```

BList {
    boolean value;
    BList queue;

    BList() {
        value := true;
        queue := null;
    }

    BList(boolean v, BList q) {
        value := v;
        queue := q;
    }
}

```

```

    BList getQueue() { return queue; }

    void setQueue(BList q) {
        queue := q;
    }

    boolean getValue() { return value; }

    ...
}

```

3. Semantics of AOO

In this section, a pointer graph semantics of AOO programs is provided. Pointer graphs are reminiscent from Cook and Rackoff's Jumping Automata on Graphs [8]. A pointer graph is basically a multigraph structure representing the memory heap, whose nodes are references. The pointer graph semantics is designed to work on such a structure together with a stack, for method calls. The semantics is a high-level semantics whose purpose is to be independent from the bytecode or low-level semantics and will be defined on meta-instructions, a meta-language of flattened instructions with stack operations.

3.1. Pointer graph

A *pointer graph* \mathcal{H} is a directed multigraph (V, A) . The nodes in V are memory references and the arrows in A link one reference to a reference of one of its fields. Nodes are labeled by class names and arrows are labeled by the field name. In what follows, let l_V be the node label mapping from V to \mathbb{C} and l_A be the arrow label mapping from A to \mathcal{F} .

The memory heap used by a AOO program will be represented by a pointer graph. This pointer graph explicits the arborescent nature of objects: each constructor call will create a new node (memory reference) of the multigraph and arrows to the nodes (memory references) of its fields. Those arrows will be annotated by the field name. The heap in which the objects are stored corresponds to this multigraph. Consequently, bounding the heap memory use consists in bounding the size of the computed multigraph.

3.2. Pointer mapping

A variable is of primitive (resp. reference) data type if it is declared using a type metavariable in \mathbb{T} (resp. \mathbb{C}). A *pointer mapping* with respect to a given pointer graph $\mathcal{H} = (V, A)$ is a partial mapping $p_{\mathcal{H}} : \mathbb{V} \cup \{\mathbf{this}\} \mapsto V \cup \text{dom}(\mathbb{T})$ associating primitive value in $\text{dom}(\mathbb{T})$ to some variable of primitive data type in \mathbb{V} and a memory reference in V to some variable of reference data type or to the current object \mathbf{this} .

As usual, the domain of a pointer mapping $p_{\mathcal{H}}$ is denoted $\text{dom}(p_{\mathcal{H}})$.

By completion, for a given variable $x \notin \text{dom}(p_{\mathcal{H}})$, let $p_{\mathcal{H}}(x) = \mathbf{null}$, if x is of type \mathbb{C} , $p_{\mathcal{H}}(x) = 0$, if x is of type \mathbf{int} , $p_{\mathcal{H}}(x) = \mathbf{false}$, if x is of type $\mathbf{boolean}$, ...

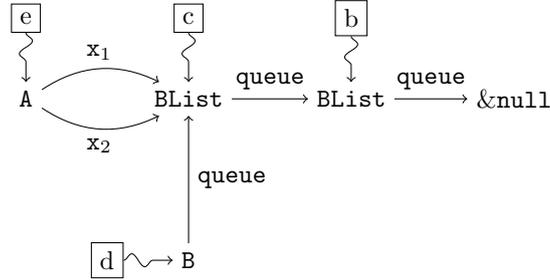


Figure 2: Example of pointer graph and pointer mapping

The use of completion is just here to ensure that the presented semantics will not get stuck.

Example 2. Consider the class `BList` of Example 1. Let `A` be a class having two `BList` fields `x1` and `x2` and `B` be a class extending `BList`. Consider the initialization instruction `Init` defined by:

```

Init ::=      BList b := new BList();
              BList c := new BList(true, b);
              B d := new B(c);
              A e := new A(c, c);

```

Figure 2 illustrates the pointer graph associated to this sequence of object creations. The figure contains both the pointer graph of labeled nodes and arrows together with the pointer mapping whose domain is represented by boxed variables and whose application is symbolized by snake arrows.

As we will see shortly, the semantics of an assignment `x := e`, for some variable `x` of reference type, consists in updating the pointer mapping in such a way that $p_{\mathcal{H}}(x)$ will be the reference of the object computed by `e`. By abuse of notation, let $p_{\mathcal{H}}(e)$ be a notation for representing this latter reference.

In what follows, let $p_{\mathcal{H}}[x \mapsto v]$, $v \in V \cup \text{dom}(\mathbb{T})$, be a notation for the pointer mapping $p'_{\mathcal{H}}$ that is equal to $p_{\mathcal{H}}$ but on `x` where the value is updated to `v`.

3.3. Pointer stack

For a given pointer graph \mathcal{H} , a *stack frame* $s_{\mathcal{H}}$ is a pair $\langle s, p_{\mathcal{H}} \rangle$ composed by a method signature $s = \tau \text{ m}^c(\tau_1, \dots, \tau_n)$ and a pointer mapping $p_{\mathcal{H}}$.

A *pointer stack* $\mathcal{S}_{\mathcal{H}}$ is a LIFO structure of stack frames corresponding to the same pointer graph \mathcal{H} . Define $\top \mathcal{S}_{\mathcal{H}}$ to be the top pointer mapping of $\mathcal{S}_{\mathcal{H}}$. Finally, define $\text{pop}(\mathcal{S}_{\mathcal{H}})$ to be the pointer stack obtained from $\mathcal{S}_{\mathcal{H}}$ by removing $\top \mathcal{S}_{\mathcal{H}}$ and $\text{push}(s_{\mathcal{H}}, \mathcal{S}_{\mathcal{H}})$ to be the pointer stack obtained by adding $s_{\mathcal{H}}$ to the top of $\mathcal{S}_{\mathcal{H}}$.

In what follows, we will write just `pop` and `push($s_{\mathcal{H}}$)` instead of `pop($\mathcal{S}_{\mathcal{H}}$)` and `push($s_{\mathcal{H}}, \mathcal{S}_{\mathcal{H}}$)` when $\mathcal{S}_{\mathcal{H}}$ is clear from the context.

As expected, the pointer stack of a program is used when calling a method: references to the parameters are pushed on a new stack frame at the top of the

pointer stack. The pointer mappings of a pointer stack $\mathcal{S}_{\mathcal{H}}$ map method parameters to the references of the arguments on which they are applied, respecting the dynamic binding principle found in Object Oriented Languages.

Example 3. For example, considering the method `setQueue(BList q)` defined in the class `BList` of Example 1, adding a method call `d.setQueue(b)`; at the end of the initialization instruction of Example 2 will push a new stack frame $\langle \text{void setQueue}^{\text{BList}}(\text{BList}), p_{\mathcal{H}} \rangle$ on the pointer stack. $p_{\mathcal{H}}(\mathbf{q})$ will point to $p_{\mathcal{H}}(\mathbf{b})$, the node corresponding to the object computed by `b`, and $p_{\mathcal{H}}(\mathbf{this})$ will point to $p_{\mathcal{H}}(\mathbf{d})$, the node corresponding to the object computed by `d`.

At the beginning of an execution, the pointer stack will only contain the stack frame $\langle \text{void main}^{\text{Exe}}(), p_0 \rangle$; p_0 being a mapping associating each local variable in the main method to the `null` reference, whether it is of reference type, and to the basic primitive value otherwise (`false` for `boolean` and `0` for `int`). We will see shortly that a pop operation removing the top pointer mapping from the pointer stack will correspond, as expected, to the evaluation of a return statement in a method body.

3.4. Memory configuration

A memory configuration $\mathcal{C} = \langle \mathcal{H}, \mathcal{S}_{\mathcal{H}} \rangle$ consists in a pointer graph \mathcal{H} together with a pointer stack $\mathcal{S}_{\mathcal{H}}$. Among memory configurations, we distinguish the *initial configuration* \mathcal{C}_0 defined by $\mathcal{C}_0 = \langle \langle \{\&\text{null}\}, \emptyset \rangle, [\langle \text{void main}^{\text{Exe}}(), p_0 \rangle] \rangle$ where `&null` is the reference of the `null` object (i.e. $l_V(\&\text{null}) = \text{null}$) and $[s_{\mathcal{H}}]$ denotes the pointer stack composed of only one stack frame $s_{\mathcal{H}}$.

In other words, the initial configuration is such that the pointer graph only contains the null reference as node and no arrows and a pointer stack with one frame for the `main` method call.

3.5. Meta-language and flattening

The semantics of AOO programs will be defined on a meta-language of expressions and instructions. Meta-expressions are flat expressions. Meta-instructions consist in flattened instructions and `pop` and `push` operations for managing method calls. Meta-expressions and meta-instructions are defined formally by the following grammar:

$$\begin{aligned}
 \text{me} & ::= \text{x} \mid \text{cst}_{\tau} \mid \text{null} \mid \text{this} \mid \text{op}(\bar{\text{x}}) \\
 & \quad \mid \text{new } \mathbf{C}(\bar{\text{x}}) \mid \text{y.m}(\bar{\text{x}}) \\
 \text{MI} & ::= ; \mid [\tau] \text{x} := \text{me}; \mid \text{MI}_1 \text{MI}_2 \mid \text{y.m}(\bar{\text{x}}); \\
 & \quad \mid \text{while}(\text{x})\{\text{MI}\} \mid \text{if}(\text{x})\{\text{MI}_1\}\text{else}\{\text{MI}_2\} \\
 & \quad \mid \text{pop}; \mid \text{push}(s_{\mathcal{H}}); \mid \epsilon
 \end{aligned}$$

where ϵ denotes the empty meta-instruction.

Flattening an instruction \mathbf{I} into a meta-instruction $\underline{\mathbf{I}}$ will consist in adding fresh intermediate variables for each complex expression parameter. This procedure is standard and defined in Figure 3. The flattened meta-instruction

$$\begin{aligned}
& \llbracket [\tau] \text{ x} := \text{e} \rrbracket = \llbracket [\tau] \text{ x} := \text{e} \rrbracket; \quad \text{if } \text{e} \in \mathbb{V} \cup \text{dom}(\mathbb{T}) \cup \{\text{this}, \text{null}\} \\
& \llbracket [\tau] \text{ x} := \text{op}(\text{e}_1, \dots, \text{e}_n) \rrbracket = \llbracket \tau_1 \text{ x}_1 := \text{e}_1; \dots \tau_n \text{ x}_n := \text{e}_n; [\tau] \text{ x} = \text{op}(\text{x}_1, \dots, \text{x}_n) \rrbracket; \\
& \llbracket [\tau] \text{ x} := \text{new } \text{C}(\text{e}_1, \dots, \text{e}_n) \rrbracket = \llbracket \tau_1 \text{ x}_1 := \text{e}_1; \dots \tau_n \text{ x}_n := \text{e}_n; [\tau] \text{ x} := \text{new } \text{C}(\text{x}_1, \dots, \text{x}_n) \rrbracket; \\
& \llbracket [\tau] \text{ x} := \text{e.m}(\text{e}_1, \dots, \text{e}_n) \rrbracket = \llbracket \tau_{n+1} \text{ x}_{n+1} = \text{e}; \tau_1 \text{ x}_1 := \text{e}_1; \dots \tau_n \text{ x}_n := \text{e}_n; \\
& \quad \llbracket [\tau] \text{ x} := \text{x}_{n+1}.\text{m}(\text{x}_1, \dots, \text{x}_n) \rrbracket; \\
& \quad \underline{\text{I}}_1 \underline{\text{I}}_2 = \underline{\text{I}}_1 \underline{\text{I}}_2 \\
& \quad \text{while}(\text{e})\{\underline{\text{I}}\} = \text{boolean } \text{x}_1 := \text{e}; \text{while}(\text{x}_1)\{\underline{\text{I}} \text{ x}_1 := \text{e};\} \\
& \quad \text{if}(\text{e})\{\underline{\text{I}}_1\}\text{else}\{\underline{\text{I}}_2\} = \text{boolean } \text{x}_1 := \text{e}; \text{if}(\text{x}_1)\{\underline{\text{I}}_1\}\text{else}\{\underline{\text{I}}_2\}
\end{aligned}$$

All x_i represent fresh variables and the types τ_i match the expressions e_i types

Figure 3: Instruction flattening

will keep the semantics of the initial instruction unchanged. The main interest in such a program transformation is just that all the variables will be statically defined in a meta-instruction whereas they could be dynamically created by an instruction, hence allowing a cleaner (and easier) semantic treatment of meta-instructions. We extend the flattening to methods (and constructors) by $\tau \text{ m}(\tau_1 \text{ x}_1, \dots, \tau_n \text{ x}_n) \{ \underline{\text{I}} [\text{return } \text{x};] \}$ so that each instruction is flattened. A flattened program \underline{P} is the program obtained by flattening all the instructions in the methods of a program P . Notice that the flattening of an AOO program is also an AOO program, as the flattening is a closed transformation with respect to the AOO syntax, and that the flattening is a polynomially bounded program transformation.

Lemma 1. *Define the size of an instruction $|\text{I}|$ (respectively meta-instruction $|\text{MI}|$) to be the number of symbols in I (resp. MI). For each instruction I , we have $|\underline{\text{I}}| = O(|\text{I}|^2)$.*

Proof. By induction on the definition of flattening. The flattening of each atomic instruction adds a number of new symbols that is at most linear in the size of the original instruction. The number of instructions is also linear in $|\text{I}|$. \square

Corollary 1. *Define the size of an AOO program $|P|$ to be the number of symbols in P . For each program P , we have $|\underline{P}| = O(|P|^2)$.*

An alternative choice would have been to restrict program syntax by requiring expressions to be flattened, thus avoiding the use of the meta-language. However such a choice would impact negatively the expressivity of this study. On another hand, one possibility might have been to generate local variables dynamically in the programming semantics but such a treatment makes the analysis of pointer mapping domain (i.e. the number of living variables) a very hard task to handle.

Example 4. We add the method `decrement()` to the class `BList` of Example 1:

```
void decrement() {
  if (value) {
    value := false;
  }
  else{
    if (queue != null) {
      value := true;
      queue.decrement();
    } else {
      value := false;
    }
  }
}
```

The program flattening will generate the following body for the flattened method:

```
void decrement() {
  if (value) {
    value := false;
  }
  else{
    BList x1 := null;
    BList x2 := queue;
    boolean x3 := x2 != x1
    if (x3) {
      value := true;
      queue.decrement();
    } else {
      value := false;
    }
  }
}
```

where x_1 , x_2 and x_3 are fresh variables.

3.6. Program semantics

Informally, the small step semantics \rightarrow of AOO programs relates a pair (\mathcal{C}, MI) of memory configuration \mathcal{C} and meta-instruction MI to another pair $(\mathcal{C}', \text{MI}')$ consisting of a new memory configuration \mathcal{C}' and of the next meta-instruction MI' to be executed. Let \rightarrow^* (respectively \rightarrow^+) be its reflexive and transitive (respectively transitive) closure. In the special case where $(\mathcal{C}, \text{MI}) \rightarrow^* (\mathcal{C}', \epsilon)$, we say that MI *terminates on memory configuration* \mathcal{C} .

Now, before defining the formal semantics of AOO programs, we introduce some preliminary notations.

Given a memory configuration $\mathcal{C} = \langle \mathcal{H}, \text{push}(\langle \tau \text{ m}^{\mathcal{C}}(\bar{\tau}), p_{\mathcal{H}} \rangle), \mathcal{S}_{\mathcal{H}} \rangle$:

- $\top \mathcal{C} = p_{\mathcal{H}}$

$$(\mathcal{C}, [\tau] \text{x}:=\text{y}; \text{MI}) \rightarrow (\mathcal{C}[\text{x} \mapsto \mathcal{C}(\text{y})], \text{MI}) \quad \text{if } \text{x} \notin \mathbb{C}(\mathcal{C}).\mathcal{F} \quad (1)$$

$$(\mathcal{C}, [\tau] \text{x}:=\text{y}; \text{MI}) \rightarrow (\mathcal{C}[\mathcal{C}(\text{this}) \xrightarrow{\text{x}} \mathcal{C}(\text{y})], \text{MI}) \quad \text{if } \begin{cases} \text{x} \in \mathbb{C}(\mathcal{C}).\mathcal{F} \\ \mathcal{C}(\text{this}) \neq \text{null} \end{cases} \quad (2)$$

$$(\mathcal{C}, [\tau] \text{x}:=\text{y}; \text{MI}) \rightarrow (\mathcal{C}, \text{MI}) \quad \text{if } \text{x} \in \mathbb{C}(\mathcal{C}).\mathcal{F} \text{ and } \mathcal{C}(\text{this}) = \text{null} \quad (3)$$

$$(\mathcal{C}, [\tau] \text{x}:=\text{cst}_\tau; \text{MI}) \rightarrow (\mathcal{C}[\text{x} \mapsto \text{cst}_\tau], \text{MI}) \quad (4)$$

$$(\mathcal{C}, [\tau] \text{x}:=\text{null}; \text{MI}) \rightarrow (\mathcal{C}[\text{x} \mapsto \&\text{null}], \text{MI}) \quad (5)$$

$$(\mathcal{C}, [\tau] \text{x}:=\text{this}; \text{MI}) \rightarrow (\mathcal{C}[\text{x} \mapsto \mathcal{C}(\text{this})], \text{MI}) \quad (6)$$

$$(\mathcal{C}, [\tau] \text{x}:=\text{op}(\bar{\text{y}}); \text{MI}) \rightarrow (\mathcal{C}[\text{x} \mapsto \llbracket \text{op} \rrbracket(\overline{\mathcal{C}(\bar{\text{y}})})], \text{MI}) \quad (7)$$

$$(\mathcal{C}, [\tau] \text{x}:=\text{new } \mathbb{C}(\bar{\text{y}}); \text{MI}) \rightarrow (\mathcal{C}[v \mapsto \mathcal{C}][v \xrightarrow{\text{z}} \mathcal{C}(\bar{\text{y}}_i)][\text{x} \mapsto v], \text{MI}) \quad (8)$$

where v is a fresh node (memory reference) and $\mathbb{C}.\mathcal{F} = \{\mathbf{z}_1, \dots, \mathbf{z}_n\}$

$$(\mathcal{C}, ; \text{MI}) \rightarrow (\mathcal{C}, \text{MI}) \quad (9)$$

$$(\mathcal{C}, [\text{u}:=]\text{x.m}(\bar{\text{y}}); \text{MI}) \rightarrow (\mathcal{C}, \quad (10)$$

$\text{push}(\langle \tau \text{ m}^{\mathbb{C}^*}(\bar{\tau}), \top \mathcal{C}[\text{this} \mapsto \mathcal{C}(\text{x}), \mathbf{z}_i \mapsto \mathcal{C}(\text{x.z}_i), \mathbf{x}_i \mapsto \mathcal{C}(\bar{\text{y}}_i)] \rangle);$

$\text{MI}' [\text{u}:=\text{x}';] \text{pop}; \text{MI}$

if m is a flattened method $\tau \text{ m}(\tau_1 \text{ x}_1, \dots, \tau_n \text{ x}_n)\{\text{MI}' [\text{return } \text{x}'];\}$ in \mathbb{C}^*

and $\mathbb{C}.\mathcal{F} = \{\mathbf{z}_1, \dots, \mathbf{z}_n\}$

$$(\mathcal{C}, \text{push}(s_{\mathcal{H}}); \text{MI}) \rightarrow (\mathcal{C}[\text{push}(s_{\mathcal{H}})], \text{MI}) \quad (11)$$

$$(\mathcal{C}, \text{pop}; \text{MI}) \rightarrow (\mathcal{C}[\text{pop}], \text{MI}) \quad (12)$$

$$(\mathcal{C}, \text{while}(\text{x})\{\text{MI}'\} \text{MI}) \rightarrow (\mathcal{C}, \text{MI}' \text{ while}(\text{x})\{\text{MI}'\} \text{MI}) \quad \text{if } \mathcal{C}(\text{x}) = \text{true} \quad (13)$$

$$(\mathcal{C}, \text{while}(\text{x})\{\text{MI}'\} \text{MI}) \rightarrow (\mathcal{C}, \text{MI}) \quad \text{if } \mathcal{C}(\text{x}) = \text{false} \quad (14)$$

$$(\mathcal{C}, \text{if}(\text{x})\{\text{MI}_{\text{true}}\}\text{else}\{\text{MI}_{\text{false}}\} \text{MI}) \rightarrow (\mathcal{C}, \text{MI}_{\mathcal{C}(\text{x})} \text{MI}) \quad (15)$$

Figure 4: Semantics of AOO programs

- $\mathcal{C}(\text{x}) = p_{\mathcal{H}}(\text{x})$
- $\mathcal{C}(\text{x.z}) = v$, with $v \in \mathcal{H}$ such that $\mathcal{C}(\text{x}) \xrightarrow{\text{z}} v \in \mathcal{H}$
- $\mathbb{C}(\mathcal{C}) = \mathbb{C}$
- $\mathcal{C}[\text{x} \mapsto v] = \langle \mathcal{H}, \text{push}(\langle \tau \text{ m}^{\mathbb{C}}(\bar{\tau}), p_{\mathcal{H}}[\text{x} \mapsto v] \rangle, \mathcal{S}_{\mathcal{H}}) \rangle$, $v \in V \cup \text{dom}(\mathbb{T})$
- $\mathcal{C}[\text{push}(s_{\mathcal{H}})] = \langle \mathcal{H}, \text{push}(s_{\mathcal{H}}, \text{push}(\langle \tau \text{ m}^{\mathbb{C}}(\bar{\tau}), p_{\mathcal{H}} \rangle, \mathcal{S}_{\mathcal{H}})) \rangle$
- $\mathcal{C}[\text{pop}] = \langle \mathcal{H}, \mathcal{S}_{\mathcal{H}} \rangle$

In other words, $\top \mathcal{C}$ is a shorthand for the pointer mapping at the top of the stack $\text{push}(\langle \tau \text{ m}^{\mathbb{C}}(\bar{\tau}), p_{\mathcal{H}} \rangle, \mathcal{S}_{\mathcal{H}})$. $\mathcal{C}(\text{x})$ is a shorthand notation for $\top \mathcal{C}(\text{x})$. $\mathcal{C}(\text{x.z})$ is the memory reference of the field \mathbf{z} of the object stored in x . $\mathbb{C}(\mathcal{C})$ is the class

of the current object under evaluation in the memory configuration. $\mathcal{C}[\mathbf{x} \mapsto v]$, $v \in V \cup \text{dom}(\mathbb{T})$, is a notation for the memory configuration \mathcal{C}' that is equal to \mathcal{C} but on $\top\mathcal{C}(\mathbf{x})$ where the value is updated to v . $\mathcal{C}[\text{push}(s_{\mathcal{H}})]$ and $\mathcal{C}[\text{pop}]$ are notations for the memory configuration where the frame $s_{\mathcal{H}}$ has been pushed to the top of the pointer stack and where the top pointer mapping has been removed from the top of the stack, respectively.

Finally, let $\mathcal{C}[v \mapsto \mathbf{C}]$, $v \in V$ denote a memory configuration \mathcal{C}' whose graph contains the new node v labeled by \mathbf{C} (i.e. $l_V(v) = \mathbf{C}$) and let $\mathcal{C}[v \overset{\mathbf{x}}{\mapsto} w]$, $v, w \in V$, denote a memory configuration \mathcal{C}' whose pointer graph contains the new arrow (v, w) labeled by \mathbf{x} (i.e. $l_A((v, w)) = \mathbf{x}$). In the case where there was already some arrow (v, u) labeled by \mathbf{x} (i.e. $l_A((v, u)) = \mathbf{x}$) in the graph then it is deleted and replaced by (v, w) .

Each operator $\text{op} \in \mathbb{O}$ of signature $\text{op} :: \tau_1 \times \dots \times \tau_n \rightarrow \tau_{n+1}$ is assumed to compute a total function $\llbracket \text{op} \rrbracket : \tau_1 \times \dots \times \tau_n \rightarrow \tau_{n+1}$ fixed by the language implementation. Operators will be discussed in more details in Subsection 4.2.

The formal rules of the small step semantics are provided in Figure 4. Let us explain the meaning of these rules.

- Rules (1) to (7) are fairly standard rules only updating the top pointer mapping $\top\mathcal{C}$ of the considered configuration but do not alter the other parts of the pointer stack. Rule (2) is the only rule altering the pointer graph. Rule (1) describes the assignment of a variable to another (that is not a field of the current object). Rule (2) describes a field assignment when the object is not null. Notice that in this case, the pointer graph is updated as the reference node of the current object points to the reference node of the assigned variable. Rule (3) indicates that if the object is null, assigning its fields results in skipping the instruction. Rule (4) is the assignment of a primitive constant to a variable. Rule (5) is the assignment of the null reference `&null` to a variable. Rule (6) consists in the assignment of the self-reference. Notice that such an assignment may only occur in a method body (because of well-formedness assumptions) and consequently the stack is non-empty and must contain a reference to `this`. Rule (7) describes operator evaluation based on the prior knowledge of function $\llbracket \text{op} \rrbracket$.
- Rule (8) consists in the creation of a new instance. Consequently, this rule adds a new node v of label \mathbf{C} and the corresponding arrows $(v, \mathcal{C}(\mathbf{y}_i))$ of label \mathbf{z}_i in the pointer graph of \mathcal{C} . $\mathcal{C}(\mathbf{y}_i)$ are the nodes of the graph (or the primitive values) corresponding to the parameters of the constructor call and \mathbf{z}_i is the corresponding field name in the class \mathbf{C} . Finally, this rule adds a link from the variable \mathbf{x} to the new reference v in the pointer mapping $\top\mathcal{C}$.
- Rule (9) just consists in the evaluation of the empty instruction `';`.
- Rule (10) consists in a call to method \mathbf{m} of class \mathbf{C} , provided that \mathbf{z} is of class \mathbf{C} , dynamically for overrides. If \mathbf{m} is not defined in \mathbf{C} then it checks

for a method of the same signature in the upper class D , and so on. Let $C\star$ be a notation for the least superclass of C in which m is defined. It adds a new instruction for pushing a new stack frame on the stack, containing references of the current object `this` on which m is applied, references of each field, and references of the parameters. After adding the flattened body MI' of m to the evaluated instruction, it adds an assignment storing the returned value z' in the assigned variable x , whenever the method is not a procedure, and a `pop`; instruction. The `pop` instruction is crucial to indicate the end of the method body so that the callee knows when to return the control flow to the caller.

- Rules (11) and (12) are standard rules for manipulating the pointer stack through the use of `pop` and `push` instructions.
- Rules (13) to (15) are standard rules for control flow statements.

One important point to stress is that, contrarily to usual OO languages, the above semantics can reduce when a method is called on a variable pointing to the value (see Rule (10)). In this particular case, all assignments to the fields of the current object are ignored (see Rule (3)). This choice has been made to simplify the formal treatment as exceptions are not included in AOO syntax. In general, exception handling generate a new control flow that is tedious to handle with a small step semantics. More, our typing discipline relies on the control flow being smooth.

3.7. Input and Size

In order to analyze the complexity of programs, it is important to relate the execution of a given program to the size of its input. Consequently, we need to define both the notion of input of an AOO program and the notion of size for such an input. This will make clear the choice made in Section 2 to express an executable class by splitting the instruction of the `main` method in an *initialization instruction* `Init` and a *computational instruction* `Comp`.

An AOO program of executable `Exe{void main(){Init Comp}}` terminates if the following conditions hold:

1. $(C_0, \underline{\text{Init}}) \rightarrow^* (\mathcal{I}, \epsilon)$
2. $(\mathcal{I}, \underline{\text{Comp}}) \rightarrow^* (C, \epsilon)$

The memory configuration \mathcal{I} computed by the initialization instruction is called the *input*.

An important point to stress is that, given a program, the choice of initialization and computational instructions is left to the analyzer. This choice is crucial for this analysis to be relevant. Note also that the initialization instruction `Init` can be changed so that the program may be defined on any input.

Another way to handle the notion of input could be to allow i/o (open a stream on a file, deserialization,...) in this instruction but at the price of a more technical and not that interesting treatment. A last possibility is to consider the input to be the main method arguments. But this would mean that as most of

the programs do not use their prompt line argument, they would be considered to be constant time programs.

There are two particular cases:

- If the initialization instruction is empty then there will be no computation on reference type variables apart from constant time or non-terminating ones, as we will see shortly. This behavior is highly expected as it means that the program has no input. As there is no input, it means that either the program does not terminate or it terminates in constant time.
- If the computational instruction is empty (that is “;”) then the program will trivially pass the complexity analysis.

Example 5. Consider the initialization instruction `Init` of Example 2. `Init` is already flattened so that $\underline{\text{Init}} = \text{Init}$ holds. The input \mathcal{I} is a memory configuration $\langle \mathcal{H}, \mathcal{S}_{\mathcal{H}} \rangle$ such that \mathcal{H} is the pointer graph described in Figure 2 and $\mathcal{S}_{\mathcal{H}} = [\langle \text{void main}^{\text{Exe}}(), p_{\mathcal{H}} \rangle]$, $p_{\mathcal{H}}$ being the pointer mapping described by the snake arrows of Figure 2.

Now consider the computational instruction `Comp = d.setQueue(b);`. It is also flat and so is the method body. Consequently, we have:

$$\begin{aligned}
(\mathcal{I}, \text{d.setQueue}(\mathbf{b});) &\rightarrow (\mathcal{I}, \text{push}(s_{\mathcal{H}}); \text{queue}:=\mathbf{q}; \text{pop};) \\
&\rightarrow (\mathcal{I}[\text{push}(s_{\mathcal{H}})], \text{queue}:=\mathbf{q}; \text{pop};) \\
&\rightarrow (\mathcal{I}'[\mathcal{I}'(\text{this}) \xrightarrow{\text{queue}} \mathcal{I}'(\mathbf{q})], \text{pop};) \\
&= (\mathcal{I}'[\mathcal{I}(\mathbf{d}) \xrightarrow{\text{queue}} \mathcal{I}(\mathbf{b})], \text{pop};) \\
&\rightarrow (\mathcal{I}'[\mathcal{I}(\mathbf{d}) \xrightarrow{\text{queue}} \mathcal{I}(\mathbf{b})][\text{pop}], \epsilon) \\
&= (\mathcal{I}[\mathcal{I}(\mathbf{d}) \xrightarrow{\text{queue}} \mathcal{I}(\mathbf{b})], \epsilon)
\end{aligned}$$

with $s_{\mathcal{H}} = \langle \text{void setQueue}^{\text{BList}}(\text{BList}), \top \mathcal{I}[\text{this} \mapsto \mathcal{I}(\mathbf{d}), \mathbf{q} \mapsto \mathcal{I}(\mathbf{b})] \rangle$ and $\mathcal{I}' = \mathcal{I}[\text{push}(s_{\mathcal{H}})]$. The two first transitions are an application of Rules (8) and (9) of Figure 4. The third one is Rule (14) as `queue` is a field of the current object. The first equality holds by definition of \mathcal{I}' and its top stack frame $\top \mathcal{I}' = s_{\mathcal{H}}$. The last reduction is Rule (10) and the equality holds as removing the top stack of \mathcal{I}' leads to \mathcal{I} , keeping in mind that the pointer graph has been updated by $[\mathcal{I}(\mathbf{d}) \xrightarrow{\text{queue}} \mathcal{I}(\mathbf{b})]$. Indeed now the node referenced by `d` points to `b` via the arrow labeled by `queue`.

Definition 1 (Sizes). The size:

1. $|\mathcal{H}|$ of a pointer graph $\mathcal{H} = (V, A)$ is defined to be the number of nodes in V .
2. $|p_{\mathcal{H}}|$ of a pointer mapping $p_{\mathcal{H}}$ is defined by $|p_{\mathcal{H}}| = \sum_{x \in \text{dom}(p_{\mathcal{H}})} |p_{\mathcal{H}}(x)|$, where the size of numerical primitive value is the value itself, the size of a boolean is 1 and the size of a memory reference is 1.
3. $|s_{\mathcal{H}}|$ of a stack frame $s_{\mathcal{H}} = \langle s, p \rangle$ is defined by: $|s_{\mathcal{H}}| = 1 + |p|$.
4. $|\mathcal{S}_{\mathcal{H}}|$ of a stack $\mathcal{S}_{\mathcal{H}}$ is defined by $|\mathcal{S}_{\mathcal{H}}| = \sum_{s_{\mathcal{H}} \in \mathcal{S}_{\mathcal{H}}} |s_{\mathcal{H}}|$.

5. $|\mathcal{C}|$ of a memory configuration $\mathcal{C} = \langle \mathcal{H}, \mathcal{S}_{\mathcal{H}} \rangle$ is defined by $|\mathcal{C}| = |\mathcal{H}| + |\mathcal{S}_{\mathcal{H}}|$.

In Item 1, it suffices to bound the number of nodes as, for the considered multigraphs, $|A| = \mathcal{O}(|V|)$. Indeed, each node has a number of out arrows bounded by a constant (the maximal number of fields in all classes of a given program). In Item 2, numerical primitive values are not considered to be constant. This definition is robust if we consider their size to be constant: e.g. a signed 32 bit integer could be considered as a constant smaller than $2^{31} - 1$. In such a case, the size of each pointer mapping would be constant as no fresh variable can be generated within a program thanks to flattening. In Item 4, the size of a pointer stack is very close to the size of the usual OO Virtual Machine stack since it counts the number of nested method calls (i.e. the number of stack frames in the stack) and the size of primitive data in each frame (that are duplicated during the pass-by-value evaluation). Finally, Item 5 defines the size of a memory configuration, the program input and bounds both the heap size of the input \mathcal{I} and *a fortiori* the stack size as the stack is empty in \mathcal{I} .

Negative integers are not considered for simplicity reasons (see section 4.2 where we need data types to have a lower bound). Floats are not considered as there is an infinite number of floats of the same size. Only finite types and countable types can be treated. To our knowledge this point is not tackled by the related works on ICC.

3.8. Compatible pairs

Given a memory configuration \mathcal{C} and a meta-instruction MI , the pair (\mathcal{C}, MI) is compatible if there exists an instruction MI' from a well-formed program such that $(\mathcal{C}_0, \text{MI}') \rightarrow^* (\mathcal{C}, \text{MI})$. Throughout the paper, we will only consider compatible pairs (\mathcal{C}, MI) .

This notion is introduced in order to prevent the consideration of a pair (\mathcal{C}, MI) having a variable not defined in the memory configuration \mathcal{C} and called without being declared in the meta-instruction MI .

Note that the semantics of Section 3.6 cannot reach incompatible pairs as each variable is supposed to be declared before its first use by definition of well-formed programs. However, this restriction will be required in order to prevent bad configurations from being considered in Theorem 1.

4. Type system

In this section, a tier based type system for ensuring polynomial time and polynomial space upper bounds on the size of a memory configuration is introduced.

4.1. Tiered types

The set of base types is defined to be the set of all primitive and reference types CUT . *Tiers* are elements of the lattice $(\{\mathbf{0}, \mathbf{1}\}, \vee, \wedge)$ where \wedge and \vee are the greatest lower bound operator and the least upper bound operator, respectively

; the induced order, denoted \preceq , being such that $\mathbf{0} \preceq \mathbf{1}$. Given a sequence of tiers $\bar{\alpha} = \alpha_1, \dots, \alpha_n$ define $\vee \bar{\alpha} = \alpha_1 \vee \dots \vee \alpha_n$. Let $\alpha, \beta, \gamma, \dots$ denote tiers in $\{\mathbf{0}, \mathbf{1}\}$.

A *tiered type* is a pair $\tau(\alpha)$ consisting of a type $\tau \in \mathbb{C} \cup \mathbb{T}$ together with a tier $\alpha \in \{\mathbf{0}, \mathbf{1}\}$.

Notations. Given two sequences of types $\bar{\tau} = \tau_1, \dots, \tau_n$ and tiers $\bar{\alpha} = \alpha_1, \dots, \alpha_n$ and a tier α , let $\bar{\tau}(\bar{\alpha})$ denote $\tau_1(\alpha_1), \dots, \tau_n(\alpha_n)$, $\bar{\tau}(\alpha)$ denote $\tau_1(\alpha), \dots, \tau_n(\alpha)$ and $\langle \bar{\tau} \rangle$ (resp. $\langle \bar{\tau}(\bar{\alpha}) \rangle$) denote the cartesian product of types (resp. tiered types).

For example, given a sequence of types $\bar{\tau} = \text{boolean}, \text{char}, \mathbb{C}$ and a sequence of tiers $\bar{\alpha} = \mathbf{0}, \mathbf{1}, \mathbf{1}$, we have $\bar{\tau}(\mathbf{0}) = \text{boolean}(\mathbf{0}), \text{char}(\mathbf{0}), \mathbb{C}(\mathbf{0})$, $\bar{\tau} = \text{boolean} \times \text{char} \times \mathbb{C}$ and $\langle \bar{\tau}(\bar{\alpha}) \rangle = \text{boolean}(\mathbf{0}) \times \text{char}(\mathbf{1}) \times \mathbb{C}(\mathbf{1})$.

What is a tier in essence. Tiers will be used to separate data in two kinds as in Bellantoni and Cook’s safe recursion scheme [1] where data are divided into “safe” and “normal” data kinds. According to Danner and Royer [9], “normal data [are the data] that drive recursions and safe data [are the data] over which recursions compute”. In this setting, tier $\mathbf{1}$ will be an equivalent for normal data type, as it consists in data that drive recursion and while loops. Tier $\mathbf{0}$ will be the equivalent for safe data type, as it consists in computational data storages. Instruction tiers will ensure that expressions of the right tier are used at the right place (e.g. tier $\mathbf{0}$ data will never drive a loop or a recursion) and that the information flow never go from $\mathbf{1}$ to $\mathbf{0}$ so that the first condition is preserved independently of tier $\mathbf{0}$ during program execution. In particular, a tier $\mathbf{1}$ instruction will never be controlled by a tier $\mathbf{0}$ instruction (in a conditional statement or recursive call).

4.2. Operators

In order to control the complexity of programs, we need to constrain the admissible tiered types of operators depending on their computational power. For that purpose and following [10], we define *neutral operators* whose computation does not make the size increase and *positive operators* whose computation may make the size increase by some constant. They are both assumed to be polynomial time computable.

Definition 2. An operator $\text{op} :: \tau_1 \times \dots \times \tau_n \rightarrow \tau$ is:

1. neutral if $\llbracket \text{op} \rrbracket$ is a polynomial time computable function and one of the following conditions hold:
 - (a) either $\tau = \text{boolean}$ or $\tau = \text{char}$,
 - (b) $\text{op} :: \text{int} \times \dots \times \text{int} \rightarrow \text{int}$ and $\forall i \in [1, n], \forall \text{cst}_{\text{int}}^i :$

$$0 \leq \llbracket \text{op} \rrbracket(\text{cst}_{\text{int}}^1, \dots, \text{cst}_{\text{int}}^n) \leq \max_j \text{cst}_{\text{int}}^j.$$

2. positive if it is not neutral, $\llbracket \text{op} \rrbracket$ is a polynomial time computable function, $\text{op} :: \text{int} \times \dots \times \text{int} \rightarrow \text{int}$, and:

$$\forall i \in [1, n], \forall \text{cst}_{\text{int}}^i, 0 \leq \llbracket \text{op} \rrbracket(\text{cst}_{\text{int}}^1, \dots, \text{cst}_{\text{int}}^n) \leq \max_j \text{cst}_{\text{int}}^j + c,$$

for some constant $c \geq 0$.

In the above definition, Item 1a could be extended to any primitive data type inhabited by a finite number of values.

In Items 1b and 2, the comparison is only defined whenever all the τ_i and τ are of type `int`. This could be extended to booleans without any trouble by considering a boolean to be 0 or 1.

Also notice that operator overload can be considered by just treating two such operators as if they were distinct ones.

Example 6. For simplicity, operators are always used in a prefix notation. The operator `-`, whose semantics is such that $\llbracket - \rrbracket(x, y) = \max(x - y, 0)$, is neutral. Indeed for all $\text{cst}_{\text{int}}^1, \text{cst}_{\text{int}}^2$, $\llbracket - \rrbracket(\text{cst}_{\text{int}}^1, \text{cst}_{\text{int}}^2) = \max(\text{cst}_{\text{int}}^1 - \text{cst}_{\text{int}}^2, 0) \leq \max(\text{cst}_{\text{int}}^1, \text{cst}_{\text{int}}^2)$. The operators `<`, testing that its left operand is strictly smaller than its right one, and the operator `==`, testing the equality of primitive values or the equality of memory references, and the operator `!=` of Example 4 are neutral operators as their output is `boolean` and they are computable in polynomial time.

The operator `+` is neither neutral, nor positive as $\llbracket + \rrbracket(\text{cst}_{\text{int}}^1, \text{cst}_{\text{int}}^2) = \text{cst}_{\text{int}}^1 + \text{cst}_{\text{int}}^2$ and there is no constant $c \geq 0$ such that $\forall \text{cst}_{\text{int}}^1, \text{cst}_{\text{int}}^2, \text{cst}_{\text{int}}^1 + \text{cst}_{\text{int}}^2 \leq \text{cst}_{\tau_i}^i + c$. However if we consider its partial evaluation `+n` to be an operator, then `+n` is a positive operator as $\forall \text{cst}_{\text{int}}^1, \llbracket +n \rrbracket(\text{cst}_{\text{int}}^1) = \text{cst}_{\text{int}}^1 + n \leq \text{cst}_{\tau_i}^i + c$. Indeed, just take $c \geq n$. The reason behind such a strange distinction is that a call of the shape `x:=y + y` could lead to an exponential computation in a loop while a call of the shape `x:=y + n` should remain polynomial (under some restrictions on the loop).

As mentioned above, the notions of neutral and positive operators can be extended to operators with numerical output and boolean input. Hence allowing to consider operators such as `x?y:z` returning `y` or `z` depending on whether `x` is `true` or `false` when applied to numerical primitive data. Indeed, we have $\llbracket ? : \rrbracket(x, y, z) \leq \max(y, z)$. Consequently, `? :` is a neutral operator.

Definition 3. An operator typing environments Ω is a mapping such that each operator $\text{op} :: \langle \bar{\tau} \rangle \rightarrow \tau$ is assigned the type:

- $\Omega(\text{op}) = \{ \langle \bar{\tau}(\mathbf{0}) \rangle \rightarrow \tau(\mathbf{0}), \langle \bar{\tau}(\mathbf{1}) \rangle \rightarrow \tau(\mathbf{1}) \}$ if `op` is a neutral operator,
- $\Omega(\text{op}) = \{ \langle \bar{\tau}(\mathbf{0}) \rangle \rightarrow \tau(\mathbf{0}) \}$ if `op` is a positive operator,
- $\Omega(\text{op}) = \emptyset$ otherwise.

4.3. Environments

In this subsection, we define three other kinds of environments:

- *method typing environments* δ associating a tiered type to each program variable $v \in \mathbb{V}$,
- *typing environments* Δ associating a typing environment δ to each method signature $\tau \mathbf{m}^c(\bar{\tau})$, i.e. $\Delta(\tau \mathbf{m}^c(\bar{\tau})) = \delta$ (by abuse of notation, we will sometime use the notation $\Delta(\mathbf{m}^c)$ when the method signature is clear from the context),
- *contextual typing environments* $\Gamma = (s, \Delta)$, a pair consisting of a method signature and a typing environment. The method (or constructor) signature s indicates under which context (typing environment) the fields are typed.

For each $\mathbf{x} \in \mathbb{V}$, define $\Gamma(\mathbf{x}) = \Delta(s)(\mathbf{x})$. Also define $\Gamma\{\mathbf{x} \leftarrow \tau(\alpha)\}$ to be the contextual typing environment Γ' such that $\forall \mathbf{y} \neq \mathbf{x}, \Gamma'(\mathbf{y}) = \Gamma(\mathbf{y})$ and $\Gamma'(\mathbf{x}) = \tau(\alpha)$. Let $\Gamma\{s'\}$ be a notation for the contextual typing environment that is equal to (s', Δ) , i.e. the signature s' has been substituted to s in Γ . Method typing environment are defined for method signatures and can be extended without any difficulty to constructor signatures.

What is the purpose of contextual environments. They will allow the type system to type a field with distinct tiered types depending on the considered method. Indeed, a field can be used in the guard of a while loop (and will be of tier **1**) in some method whereas it can store freshly created objects (and will be of tier **0**) in some other method. This is the reason why the presented type system has to keep information on the context.

4.4. Judgments

Expressions and instructions will be typed using tiered types. A method of arity n method will be given a type of the shape $\mathbb{C}(\beta) \times \tau_1(\alpha_1) \times \dots \times \tau_n(\alpha_n) \rightarrow \tau(\alpha)$. Consequently, given a contextual typing environment Γ , there are four kinds of typing judgments:

- $\Gamma, \Omega \vdash_{\beta} \mathbf{e} : \tau(\alpha)$ for expressions, meaning that the expression \mathbf{e} is of tiered type $\tau(\alpha)$ under the contextual typing environment Γ and operator typing environment Ω and can only be assigned to in an instruction of tier at least β ,
- $\Gamma, \Omega \vdash \mathbf{I} : \mathbf{void}(\alpha)$ for instructions, meaning that the instruction \mathbf{I} is of tiered type $\mathbf{void}(\alpha)$ under the contextual typing environment Γ and operator typing environment Ω ,
- $\Gamma, \Omega \vdash_{\beta} s : \mathbb{C}(\beta) \times \langle \bar{\tau}(\bar{\alpha}) \rangle \rightarrow \tau(\alpha)$ for method signatures, meaning that the method \mathbf{m} of signature s belongs to the class \mathbb{C} ($\mathbb{C}(\beta)$ is the tiered type of the current object **this**), has parameters of type $\langle \bar{\tau}(\bar{\alpha}) \rangle$, has a return variable of type $\tau(\alpha)$, with $\tau = \mathbf{void}$ in the particular case where there is

no return statement, and can only be called in instructions of tier at least β ,

- $\Gamma, \Omega \vdash \mathbf{C}(\bar{\tau}) : \langle \bar{\tau}(\mathbf{0}) \rangle \rightarrow \mathbf{C}(\mathbf{0})$ for constructor signatures, meaning that the constructor \mathbf{C} has parameters of type $\langle \bar{\tau}(\mathbf{0}) \rangle$ and a return variable of type $\mathbf{C}(\mathbf{0})$, matching the class type \mathbf{C} .

Given a sequence $\bar{\mathbf{e}} = \mathbf{e}_1, \dots, \mathbf{e}_n$ of expressions, a sequence of types $\bar{\tau} = \tau_1, \dots, \tau_n$ and two sequences of tiers $\bar{\alpha} = \alpha_1, \dots, \alpha_n$ and $\bar{\beta} = \beta_1, \dots, \beta_n$, the notation $\Gamma, \Omega \vdash_{\bar{\beta}} \bar{\mathbf{e}} : \bar{\tau}(\bar{\alpha})$ means that $\Gamma, \Omega \vdash_{\beta_i} \mathbf{e}_i : \tau_i(\alpha_i)$ holds, for all $i \in [1, n]$.

4.5. Typing rules

The intuition of the typing discipline is as follows: keeping in mind, that tier $\mathbf{1}$ corresponds to while loop guards data and that tier $\mathbf{0}$ corresponds to data storages (thus possibly increasing data), the type system precludes flows from tier $\mathbf{0}$ data to tier $\mathbf{1}$ data.

Most of the rules are basic non-interference typing rules following Volpano *et al.* type discipline: tiers in the rule premises (when there is one) are equal to the tier in the rule conclusion so that there can be no information flow (in both directions) using these rules.

4.5.1. Expressions typing rules

The typing rules for expressions are provided in Figure 5. They can be explained briefly:

- Primitive constants and the null reference can be given any tier α as they have no computational power (Rules *(Cst)* and *(Null)*) and can be used in instructions of any tier β . As in Java and for polymorphic reasons, `null` can be considered of any class \mathbf{C} .
- Rule *(Var)* just consists in checking a variable tiered type with respect to a given contextual typing environment. This is the rule allowing a polymorphic treatment of fields depending on the context. Indeed, remember that $\Gamma(\mathbf{x})$ is a shorthand notation for $\Delta(s)(\mathbf{x})$. Moreover, a variable can be used in instructions of any tier β .
- Rule *(Op)* just consists in checking that the type of an operator with respect to a given operator typing environment matches the input and output tiered types. The operator output has to be used in an instruction of tier being at least the maximal admissible tier of its operand $\bigvee \bar{\beta} = \beta_1 \vee \dots \vee \beta_n$. In other words, if one operand can only be used in a tier $\mathbf{1}$ instruction then the operator output can only be used in a tier $\mathbf{1}$ instruction.
- The rule *(Self)* makes explicit that the self reference `this` is of type \mathbf{C} and enforces the tier of the fields to be equal to the tier of the current object, thus preventing “flows by references” in the heap. Moreover it can be used in instructions of any tier β .

- The rule (*Pol*) allows us to type a given expression with the superclass type while keeping the tiers unchanged.
- The rule (*New*) checks that the constructor arguments and output all have tier $\mathbf{0}$. The new instance has to be of tier $\mathbf{0}$ since its creation makes the memory grow (a new reference node is added in the heap). The constructor arguments have also to be of tier $\mathbf{0}$. Otherwise a flow from tier $\mathbf{0}$, the new instance, to tier $\mathbf{1}$, one of its fields, might occur. The explanation for the annotation instruction tier β is the same as the one for operators.
- Rule (*C*) just checks a direct type correspondence between the arguments types and the method type when a method is called. However this rule is very important as it allows a polymorphic type discipline for fields. Indeed the contextual environment is updated so that a field can be typed with respect to the considered method. The explanation for the annotation instruction tier β is the same as the one for operators. We will see shortly how to make restriction on such annotations in order to restrict the complexity of recursive method calls.

$$\begin{array}{c}
\frac{}{\Gamma, \Omega \vdash_{\beta} \mathbf{cst}_{\tau} : \tau(\alpha)} \text{ (Cst)} \quad \frac{}{\Gamma, \Omega \vdash_{\beta} \mathbf{null} : \mathbf{C}(\alpha)} \text{ (Null)} \\
\frac{\Gamma(\mathbf{x}) = \tau(\alpha)}{\Gamma, \Omega \vdash_{\beta} \mathbf{x} : \tau(\alpha)} \text{ (Var)} \\
\frac{\Gamma, \Omega \vdash_{\beta} \bar{\mathbf{e}} : \bar{\tau}(\alpha) \quad \langle \bar{\tau}(\alpha) \rangle \rightarrow \tau(\alpha) \in \Omega(\mathbf{op})}{\Gamma, \Omega \vdash_{\sqrt{\beta}} \mathbf{op}(\bar{\mathbf{e}}) : \tau(\alpha)} \text{ (Op)} \\
\frac{\forall \mathbf{x} \in \mathbf{C.F}, \exists \tau, \Gamma(\mathbf{x}) = \tau(\alpha)}{\Gamma, \Omega \vdash_{\beta} \mathbf{this} : \mathbf{C}(\alpha)} \text{ (Self)} \quad \frac{\Gamma, \Omega \vdash_{\beta} \mathbf{e} : \mathbf{D}(\alpha) \quad \mathbf{D} \leq \mathbf{C}}{\Gamma, \Omega \vdash_{\beta} \mathbf{e} : \mathbf{C}(\alpha)} \text{ (Pol)} \\
\frac{\Gamma, \Omega \vdash_{\beta} \bar{\mathbf{e}} : \bar{\tau}(\mathbf{0}) \quad \Gamma\{\mathbf{C}(\bar{\tau})\}, \Omega \vdash \mathbf{C}(\bar{\tau}) : \langle \bar{\tau}(\mathbf{0}) \rangle \rightarrow \mathbf{C}(\mathbf{0})}{\Gamma, \Omega \vdash_{\sqrt{\beta}} \mathbf{new} \mathbf{C}(\bar{\mathbf{e}}) : \mathbf{C}(\mathbf{0})} \text{ (New)} \\
\frac{\Gamma, \Omega \vdash_{\beta} \mathbf{e} : \mathbf{C}(\beta) \quad \Gamma, \Omega \vdash_{\beta} \bar{\mathbf{e}} : \bar{\tau}(\bar{\alpha}) \quad \Gamma\{\tau \mathbf{m}^{\mathbf{C}}(\bar{\tau})\}, \Omega \vdash_{\beta} \tau \mathbf{m}^{\mathbf{C}}(\bar{\tau}) : \mathbf{C}(\beta) \times \langle \bar{\tau}(\bar{\alpha}) \rangle \rightarrow \tau(\alpha)}{\Gamma, \Omega \vdash_{\beta \vee (\sqrt{\beta})} \mathbf{e.m}(\bar{\mathbf{e}}) : \tau(\alpha)} \text{ (C)}
\end{array}$$

Figure 5: Typing rules for expressions

4.5.2. Instructions typing rules

The typing rules for instructions are provided in Figure 6. They can be explained briefly:

- Rule (*Skip*) is straightforward.
- Rule (*Seq*) shows that the tier of the sequence $\mathbf{I}_1 \mathbf{I}_2$ will be the maximum of the tiers of \mathbf{I}_1 and \mathbf{I}_2 . It can be read as “a sequence of instructions

including at least one instruction that cannot be controlled by tier $\mathbf{0}$ cannot be controlled by tier $\mathbf{0}$ ” and it preserves non-interference as it is a weakly monotonic typing rule with respect to tiers.

- The recovery is performed thanks to the Rule (*ISub*) that makes possible to type an instruction of tier $\mathbf{0}$ by $\mathbf{1}$ (as tier $\mathbf{0}$ instruction use is less constrained than tier $\mathbf{1}$ instruction use) without breaking the system non-interference properties. Notice also that there is no counterpart for expressions as a subtyping rule from $\mathbf{1}$ to $\mathbf{0}$ would allow us to type $x+1$; with x of tier $\mathbf{1}$ while a subtyping rule from $\mathbf{0}$ to $\mathbf{1}$ would allow the programmer to type programs with tier $\mathbf{0}$ variables in the guards of while loops.
- Rule (*Ass*) is an important non-interference rule. It forbids information flows of reference type from a tier to another: it is only possible to assign an expression e of tier α to a variable x of tier α . In the particular case of primitive data, a flow from $\mathbf{1}$ to $\mathbf{0}$ might occur. This is just because primitive data are supposed to be passed-by-value (otherwise we should keep the equality of tier preserved). In the case of a field assignment, all tiers are constrained to be $\mathbf{0}$ in order to avoid changes inside the tier $\mathbf{1}$ graph. Finally, if the expression e cannot be used in instructions of tier less than β , we constrain the instruction tiered type to be $\text{void}(\alpha \vee \beta)$ in order to fulfill this requirement.
- Rule (*If*) constrains the tier of the conditional guard e to match the tiers of the branching instructions I_1 and I_2 . Hence, it prevents assignments of tier $\mathbf{1}$ variables to be controlled by a tier $\mathbf{0}$ expression.
- Rule (*Wh*) constrains the guard of the loop e to be a boolean expression of tier $\mathbf{1}$, thus preventing while loops from being controlled by tier $\mathbf{0}$ expressions.

4.5.3. Methods typing rules

The typing rules for methods are provided in Figure 7. They can be explained briefly:

- Rule (*Body*) shows how to type method definitions. It updates the environment with respect to the parameters, current object and return type in order to allow a polymorphic typing discipline for methods: while typing a program, a method can be given distinct types depending on where and how it is called. The tier γ of the method body is used as annotation so that if $\gamma = \mathbf{1}$, the method cannot be called in a tier $\mathbf{0}$ instruction. We also check that the current object matches the tier β . As presented, this rule may create an infinite branch in the typing tree of a recursive method. This could be solved by keeping the set of methods previously typed in a third context, we chose not to include this complication. Consequently, a method is implicitly typed only once in a given branch of a typing derivation.

$$\begin{array}{c}
\frac{}{\Gamma, \Omega \vdash ; : \text{void}(\alpha)} \textit{Skip} \quad \frac{\forall i, \Gamma, \Omega \vdash I_i : \text{void}(\alpha_i)}{\Gamma, \Omega \vdash I_1 I_2 : \text{void}(\alpha_1 \vee \alpha_2)} \textit{Seq} \\
\frac{\Gamma, \Omega \vdash I : \text{void}(\mathbf{0})}{\Gamma, \Omega \vdash I : \text{void}(\mathbf{1})} \textit{ISub} \quad \frac{[\Gamma, \Omega \vdash_{\mathbf{0}} x : \tau(\alpha')] \quad \Gamma, \Omega \vdash_{\beta} e : \tau(\alpha)}{\Gamma, \Omega \vdash [[\tau] x :=] e ; : \text{void}(\alpha \vee \beta)} \textit{Ass}^{**} \\
\frac{\Gamma, \Omega \vdash_{\alpha} e : \text{boolean}(\alpha) \quad \forall i, \Gamma, \Omega \vdash I_i : \text{void}(\alpha)}{\Gamma, \Omega \vdash \text{if}(e)\{I_1\}\text{else}\{I_2\} : \text{void}(\alpha)} \textit{If} \\
\frac{\Gamma, \Omega \vdash_{\mathbf{1}} e : \text{boolean}(\mathbf{1}) \quad \Gamma, \Omega \vdash I : \text{void}(\mathbf{1})}{\Gamma, \Omega \vdash \text{while}(e)\{I\} : \text{void}(\mathbf{1})} \textit{Wh} \\
\textit{(**)} \\
x \in \mathcal{F} \implies \alpha = \mathbf{0} \\
\tau \in \mathbb{T} \implies \alpha' \leq \alpha \\
\tau \in \mathbb{C} \implies \alpha' = \alpha
\end{array}$$

Figure 6: Typing rules for instructions

- Rule *(Cons)* shows how to type constructors. The constructor body and parameters are enforced to be of tier $\mathbf{0}$ as a constructor invocation makes the memory grow.
- Rule *(OR)* deals with overridden method, keeping tiers preserved, thus allowing standard OO polymorphism.

$$\begin{array}{c}
\frac{\Gamma\{\text{this} \leftarrow \mathcal{C}(\beta), \bar{x} \leftarrow \bar{\tau}(\bar{\alpha}), [x \leftarrow \tau(\alpha)]\}, \Omega \vdash I : \text{void}(\gamma) \quad \Gamma, \Omega \vdash_{\gamma} \text{this} : \mathcal{C}(\beta)}{\Gamma, \Omega \vdash_{\gamma} \tau \mathfrak{m}^{\mathcal{C}}(\bar{\tau}) : \mathcal{C}(\beta) \times \langle \bar{\tau}(\bar{\alpha}) \rangle \rightarrow \tau(\alpha)} \textit{Body}^{**} \\
\frac{\Gamma\{\bar{x} \leftarrow \bar{\tau}(\mathbf{0})\}, \Omega \vdash I : \text{void}(\mathbf{0}) \quad \mathcal{C}(\bar{\tau} \bar{x})\{I\} \in \mathbb{C}}{\Gamma, \Omega \vdash \mathcal{C}(\bar{\tau}) : \langle \bar{\tau}(\mathbf{0}) \rangle \rightarrow \mathcal{C}(\mathbf{0})} \textit{Cons} \\
\frac{\mathbb{C} \triangleleft \mathbb{D} \quad \Gamma, \Omega \vdash_{\gamma} \tau \mathfrak{m}^{\mathbb{D}}(\bar{\tau}) : \mathbb{D}(\beta) \times \langle \bar{\tau}(\bar{\alpha}) \rangle \rightarrow \tau(\alpha) \quad \tau \mathfrak{m}(\bar{\tau} \bar{x})\{I [\text{return } x;]\} \in \mathbb{D}}{\Gamma, \Omega \vdash_{\gamma} \tau \mathfrak{m}^{\mathcal{C}}(\bar{\tau}) : \mathcal{C}(\beta) \times \langle \bar{\tau}(\bar{\alpha}) \rangle \rightarrow \tau(\alpha)} \textit{OR} \\
\textit{(**) provided that } \tau \mathfrak{m}(\bar{\tau} \bar{x})\{I [\text{return } x;]\} \in \mathbb{C}
\end{array}$$

Figure 7: Typing rules for methods

In the above rules, the tier γ is just an annotation on the least tier of the instructions where the method is called. It will be used in the next Section to constrain the tier of recursive methods.

4.6. Well-typedness.

Given a program of executable $\text{Exe}\{\text{void main}()\{\text{Init Comp}\}\}$, a typing environment Δ and operator typing environment Ω , the judgment:

$$\Delta, \Omega \vdash \text{Exe}\{\text{void main}()\{\text{Init Comp}\}\} : \diamond$$

means that the program is well-typed with respect to Δ and Ω and is defined by:

$$\frac{(\text{void main}^{\text{Exe}}(), \Delta), \Omega \models \text{Init} : \text{void} \quad (\text{void main}^{\text{Exe}}(), \Delta), \Omega \vdash \text{Comp} : \text{void}(1)}{(\text{void main}^{\text{Exe}}(), \Delta), \Omega \vdash \text{Exe}\{\text{void main}()\{\text{Init Comp}\}\} : \diamond}$$

where \models is a judgment derived from the type system by removing all tiers and tier based constraints in the typing rules. For example, Rule (C) of Figure 5 becomes:

$$\frac{\Gamma, \Omega \models e : C \quad \Gamma, \Omega \models \bar{e} : \bar{\tau} \quad \Gamma\{\tau \text{ m}^c(\bar{\tau})\}, \Omega \models \tau \text{ m}^c(\bar{\tau}) : C \times \langle \bar{\tau} \rangle \rightarrow \tau}{\Gamma, \Omega \models e.m(\bar{e}) : \tau} \quad (C)$$

or Rule (Body) of Figure 7 becomes:

$$\frac{\Gamma\{\text{this} \leftarrow C, \bar{x} \leftarrow \bar{\tau}, [x \leftarrow \tau]\}, \Omega \models I : \text{void} \quad \Gamma, \Omega \models \text{this} : C}{\Gamma, \Omega \models \tau \text{ m}^c(\bar{\tau}) : C \times \langle \bar{\tau} \rangle \rightarrow \tau} \quad (\text{Body})$$

It means that all the environments are lifted so that types are substituted to tiered types.

Since no tier constraint is checked in the initialization instruction **Init**, the complexity of this latter instruction is not under control ; as explained previously the main reason for this choice is that this instruction is considered to be building the program input. In contrast, the computational instruction **Comp** is considered to be the computational part of the program and has to respect the tiering discipline.

4.7. Examples

4.7.1. Well-typedness and type derivations

Example 7. Consider a program having the following computational instruction:

```

Exe {
  void main(){
    //Init
    int n := ... ;
    BList b := null;
    while(n>0){
      b := new BList(true, b);
      n := n-1;
    }
    //Comp
    int z := 0;
    while(b.getTail() != null){
      b := b.getTail();
      z := z+1;
    }
  }
}

```

It is well-typed as it can be typed by the following derivation:

$$\frac{\frac{\Gamma(z) = \text{int}(\mathbf{0})}{\Gamma, \Omega \vdash_{\mathbf{0}} z : \text{int}(\mathbf{0})} (\text{Var}) \quad \frac{}{\Gamma, \Omega \vdash_{\mathbf{0}} 0 : \text{int}(\mathbf{0})} (\text{Cst}) \quad \frac{\vdots}{(\Pi)} (\text{Wh})}{\frac{\Gamma, \Omega \vdash z := 0; : \text{void}(\mathbf{0}) \quad \Gamma, \Omega \vdash \text{I} : \text{void}(\mathbf{1})}{\Gamma, \Omega \vdash \text{Comp} : \text{void}(\mathbf{1})} (\text{Seq})} (\text{Ass})$$

where $\text{I} = \text{while}(\text{b.getQueue}()! = \text{null})\{\text{b} := \text{b.getQueue}(); \text{z} := \text{z}+1;\}$. Now we consider the sub-derivation Π , where we omit Γ, Ω in the context in order to lighten the notation and where $\text{J} = \text{b} := \text{b.getQueue}(); \text{z} := \text{z}+1; :$

$$\frac{\frac{\frac{\vdots}{\Pi_1} (\text{C}) \quad \frac{}{\vdash_{\mathbf{1}} \text{b.getQueue}() : \text{BList}(\mathbf{1})} (\text{C})}{\vdash_{\mathbf{1}} \text{b.getQueue}()! = \text{null} : \text{boolean}(\mathbf{1})} (\text{Seq}) \quad \frac{\vdots}{\Pi_2} (\text{Wh})}{\vdash_{\mathbf{1}} \text{I} : \text{void}(\mathbf{1})} (\text{Wh})$$

For Π_1 we have:

$$\frac{\frac{\Gamma(\text{b}) = \text{BList}(\mathbf{1}) \quad \frac{\{\text{getQueue}\}[\text{this}, \text{queue} : \text{BList}(\mathbf{1})], \Omega \vdash; : \text{void}(\mathbf{1}) \dots}{\vdash_{\mathbf{1}} \text{b} : \text{BList}(\mathbf{1})} (\text{C})}{\vdash_{\mathbf{1}} \text{b.getQueue}() : \text{BList}(\mathbf{1})} (\text{C})$$

where $\{\text{getQueue}\}$ is a shorthand notation for $\Gamma\{\text{BList getQueue}^{\text{BList}(\mathbf{1})}\}, \Omega$ and $[\text{this}, \text{queue} : \text{BList}(\mathbf{1})]$ is a shorthand notation for $[\text{this} \leftarrow \text{BList}(\mathbf{1}), \text{queue} \leftarrow \text{BList}(\mathbf{1})]$.

For Π_2 , provided that $\text{me} = \text{b.getQueue}()$, we have:

$$\frac{\frac{\Gamma(\text{b}) = \text{BList}(\mathbf{1})}{\vdash_{\mathbf{0}} \text{b} : \text{BList}(\mathbf{1})} \quad \frac{\Pi_1}{\vdash_{\mathbf{1}} \text{me} : \text{BList}(\mathbf{1})} \quad \frac{\Gamma(z) = \text{int}(\mathbf{0})}{\vdash_{\mathbf{0}} z : \text{int}(\mathbf{0})} \quad \frac{\vdots}{\vdash_{\mathbf{0}} z + 1 : \text{int}(\mathbf{0})}}{\frac{\vdash_{\mathbf{0}} \text{b} := \text{me}; : \text{void}(\mathbf{1}) \quad \vdash_{\mathbf{0}} z := z + 1; : \text{void}(\mathbf{0})}{\vdash_{\mathbf{1}} \text{J} : \text{void}(\mathbf{1})} (\text{Seq})$$

Though incomplete, the above derivation can be fully typed as we have already seen in Example 6 that $+1$ is a positive operator. Consequently, it can be given the type $\text{int}(\mathbf{0}) \rightarrow \text{int}(\mathbf{0})$ and $z + 1$ can be typed, provided that $\Gamma(z) = \text{int}(\mathbf{0})$.

4.7.2. Light notation: BList revisited

As we have seen above, type derivation can be tricky to provide. In order to overcome this problem, we will no longer use typing derivations: tiers will be made explicit in the code. The notation \mathbf{x}^α means that \mathbf{x} has tier α under the considered environments Γ, Ω , i.e. $\Gamma, \Omega \vdash_\beta \mathbf{x} : \tau(\alpha)$, for some τ and β , whereas $\text{I} : \alpha$ means that $\Gamma, \Omega \vdash \text{I} : \text{void}(\alpha)$.

Example 8. We will consider some of the constructors and methods of the class `BList` to illustrate this notation.

```
BList { /* List of booleans: coding binary integers */
    boolean value;
    BList queue;

    BList(boolean v, BList q) {
        value := v;
        queue := q;
    }

    BList getQueue() { return queue; }

    void setQueue(BList q) {
        queue := q;
    }

    boolean getValue() { return value; }

    BList clone() {
        BList n;
        if (value != null) {
            n = new BList(value, queue.clone());
        } else {
            n = new BList(null, null);
        }
        return n;
    }

    void decrement() {
        if (value == true) {
            value := false;
        } else {
            if (queue != null) {
                value := true;
                queue.decrement();
            } else {
                value := false;
            }
        }
    }

    int length () {
        int res := 1;
        if ( queue != null ) {
            res := queue.length();
            res := res +1;
        }
        else {};
    }
}
```

```

        return res ;
    }

    boolean isEqual(BList other) {
        boolean res := true;
        BList b1 := this;
        BList b2 := other;
        while (b1 != null && b2 != null) {
            if (b1.getValue() != b2.getValue()){
                res := false
            } else {;;}
            b1 := b1.getQueue();
            b2 := b2.getQueue();
        }
        if (b1 != null || b2 != null) {
            res := false;
        }
        return res;
    }
}

```

Let us now consider and type each method and constructor:

```

BList(boolean v0, BList q0) {
    value := v; : 0
    queue := q; : 0
}

```

The constructor `BList` can be typed by $\text{boolean}(\mathbf{0}) \times \text{BList}(\mathbf{0}) \rightarrow \text{BList}(\mathbf{0})$, using Rules (Cons) and (Seq) and twice Rule (Ass). In this latter applications, the tier are enforced to be $\mathbf{0}$ because of the side condition $x \in \mathcal{F} \implies \alpha = \mathbf{0}$. As a consequence, it is not possible to create objects of tiered type `BList(1)` in a computational instruction.

```

BList getQueue() {
    return queue;
}

```

We have already seen in Example 7 that the method `getQueue` can be typed by $\text{BList}(\mathbf{1}) \rightarrow \text{BList}(\mathbf{1})$. It can also be typed by $\text{BList}(\mathbf{0}) \rightarrow \text{BList}(\mathbf{0})$, by Rules (Body) and (Self).

The types $\text{BList}(\alpha) \rightarrow \text{BList}(\beta)$, $\alpha \neq \beta$, are prohibited because of Rules (Body) and (Self) since the tier of the current object has to match the tier of its fields.

In the same manner, the method `getValue` can be given the types $\text{BList}(\alpha) \rightarrow \text{boolean}(\alpha)$, $\alpha \in \{\mathbf{0}, \mathbf{1}\}$.

The method `setQueue`:

```

void setQueue(BList q0) {
    queue := q; : 0
}

```

can only be given the types $\text{BList}(\mathbf{0}) \times \text{BList}(\mathbf{0}) \rightarrow \text{void}(\beta)$. Indeed, by Rule (Ass), `queue` and `q` are enforced to be of tier $\mathbf{0}$. Consequently, this is of tier $\mathbf{0}$ by Rules (Body) and (Self). The tier of the body is not constrained.

Consider the method `decrement`:

```
void decrement() {
  if (value0 == true) {
    value0 := false; :0
  } else {
    if (queue0 != null) {
      value := true;
      queue0.decrement(); :0
    } else {
      value0 := false; :0
    }
  }
}
:0
```

Because of the Rules (Ass) and (Body), it can be given the type $\text{BList}(\mathbf{0}) \rightarrow \text{void}(\mathbf{0})$. As we shall see later in Subsection 5.4, this method will be rejected by the safety condition as it might lead to exponential length derivation whenever it is called in a while loop.

Now consider the following method:

```
int length() {
  int res := 1; : 0
  if (queue1 != null) {
    res := queue.length(); : 1 //using (ISub)
    res := res+1; : 0
  }
  else {};
  return res;
}
```

It can be typed by $\Gamma, \Omega \vdash_1 \text{int length}^{\text{BList}()} : \text{BList}(\mathbf{1}) \rightarrow \text{int}(\mathbf{0})$.

Now consider the method testing the equality:

```
boolean isEqual(BList other1) {
  boolean res0 := true; :1
  BList b11 := this1; :1
  BList b21 := other1; :1
  while (b11 != null && b21 != null){
    if (b11.getValue() != b21.getValue()){
      res0 := false; :1 //using (ISub)
    } else {};
    b11 := b11.getQueue(); :1
    b21 := b21.getQueue(); :1
  }
  if (b11 != null || b21 != null) {
    res0 := false; :1 //using (ISub)
  } else {}; :1
}
```

```

    return res0;
}

```

The local variables `b1` and `b2` are enforced to be of tier **1** by Rules (Wh) and (Op). Consequently, `this` and `other` are also of tier **1** using twice Rule (Ass). This is possible as it does not correspond to field assignments. Moreover, the methods `getValue` and `getQueue` will be typed by $\text{BList}(\mathbf{1}) \rightarrow \text{boolean}(\mathbf{1})$ and $\text{BList}(\mathbf{1}) \rightarrow \text{BList}(\mathbf{1})$, respectively. Finally, the local variable can be given the type `boolean(1)` or `boolean(0)` (in this latter case, the subtyping Rule (ISub) will be needed as illustrated in the above instruction) and, consequently, the admissible types for `isEqual` are $\text{BList}(\mathbf{1}) \times \text{BList}(\mathbf{1}) \rightarrow \text{boolean}(\alpha)$, $\alpha \in \{\mathbf{0}, \mathbf{1}\}$.

4.7.3. Examples of overriding

Example 9 (Override1). Consider the following classes:

```

A {
  int x;
  ...
  void f(int y){
    x := x+1; : 1 //using (ISub)
  }
}

B extends A{
  void f(int y){
    while(y1>0){
      x := x+1; : 0
      y := y-1; : 1
    }
  }
}

```

In the class `A`, the method `f` can be given the type $\text{A}(\mathbf{0}) \times \text{int}(\mathbf{1}) \rightarrow \text{void}(\mathbf{1})$ using Rules (Op), (Ass), (Self) and (ISub). In the class `B`, the method `f` can be given the type $\text{B}(\mathbf{0}) \times \text{int}(\mathbf{1}) \rightarrow \text{void}(\mathbf{1})$ using Rules (Op), (Ass), (Seq), (Wh) and (Self). Consequently, the following code can be typed:

```

A x = ...
if (condition){
  x=new A(...);
}else{
  x=new B(...);
}
x.f(25);

```

provided that `condition` is of tier **1** and using Rules (Pol) and (OR). Indeed, we are not able to predict statically which one of the two method will be called. However we know that the argument is of tier **0** as its field will increase polynomially in both cases.

Example 10 (Override2). Consider the class `BList` and a subclass `PairOfBList`:

```

public class BList{
    ...
    int length () {
        int res := 1;
        if ( queue != null ) {
            res := queue.length();
            res := res +1;
        }
        else {};
        return res ;
    }
}

public class PairOfBList extends BList {
    Blist l1;

    int length () {
        int res0 := queue1.length()0+2;
        while ( l1 != null ) {
            l1 := l1.getQueue();
            res0 := res0 +1;
        }
        else {};
        return res0 ;
    }
}

```

The override method `length` of `PairOfBList` computes the size of a pair of `BList` objects, that is the sum of their respective size. As highlighted by the annotations, it can be given the type $\text{PairOfBList}(\mathbf{1}) \rightarrow \text{int}(\mathbf{0})$. Consequently, a method call of the shape:

$$\text{int } i^0 := p.\text{length}()^0; \text{ void}(\mathbf{1})$$

p being of type `PairOfBList`, can be typed using the Rule (OR):

$$\frac{\text{PairOfBList} \triangleleft \text{BList} \quad \Gamma, \Omega \vdash_{\mathbf{1}} \text{int length}^{\text{BList}}() : \text{BList}(\mathbf{1}) \rightarrow \text{int}(\mathbf{0})}{\Gamma, \Omega \vdash_{\mathbf{1}} \text{int length}^{\text{PairOfBList}}() : \text{PairOfBList}(\mathbf{1}) \times \rightarrow \text{int}(\mathbf{0})}$$

as Example 8 has demonstrated that the judgment $\Gamma, \Omega \vdash_{\mathbf{1}} \text{int length}^{\text{BList}}() : \text{BList}(\mathbf{1}) \rightarrow \text{int}(\mathbf{0})$ can be derived. In order for the program to be safe, the tier annotation is $\mathbf{1}$ and, consequently, the assignment is of tiered type `void(1)` (hence cannot be guarded by $\mathbf{0}$ data).

4.7.4. Exponential as a counter-example

Example 11. Now we illustrate the limits of the type system with respect to the following method computing an exponential:

```

exp(int x, int y){
  while(x1>0){
    int u? := y0;
    while (u?>0){
      y := y0+1; : 0
      u := u?-1; : ?
    }
    x := x1-1;
  }
  return y0;
}

```

It is not typable in the presented formalism. Indeed, suppose that it is typable. The expression $y+1;$ enforces y to be of tier $\mathbf{0}$ by Rule (Op) and by definition of positive operators. Consequently, the instruction $\text{int } u := y;$ enforces u to be of tier $\mathbf{0}$ because of typing discipline for assignments (Rule (Ass)). However, the innermost while loop enforces $u > 0$ to be of tier $\mathbf{1}$ by Rules (Wh) and (Op), so that u has to be of tier $\mathbf{1}$ and we obtain a contradiction.

Now, one might suggest that the exponential could be computed using an intermediate method add for addition:

```

add(int x, int y){
  while(x1>0){
    x1 := x1-1; : 1
    y0 := y0+1; : 0
  }
  return y0;
}

expo(int x){
  int res := 1;
  while(x1>0){
    res := add(res?, res?);
    x := x1-1; : 1
  }
  return res;
}

```

Thankfully, this program is not typable as the first argument of add is enforced to be of tier $\mathbf{1}$ and the second argument is enforced to be of tier $\mathbf{0}$ (see previous section). Hence, the variable `res` would have two distinct tiers under the same context. That is clearly not allowed by the type system.

4.8. Type preservation under flattening

We show that the flattening of a typable instruction has a type preservation property. A direct consequence is that the flattened program can be considered instead of the initial program.

Proposition 1. *Given an instruction I , a contextual typing environment Γ and an operator typing environment Ω such that $\Gamma, \Omega \vdash I : \text{void}(\alpha)$ holds, there is a contextual typing environment Γ' such that the following holds:*

- $\forall \mathbf{x} \in I, \Gamma'(\mathbf{x}) = \Gamma(\mathbf{x})$
- $\Gamma', \Omega \vdash \underline{I} : \text{void}(\alpha)$

where $\mathbf{x} \in I$ means that the variable \mathbf{x} appears in I .

Conversely, if $\Gamma', \Omega \vdash \underline{I} : \text{void}(\alpha)$, then $\Gamma', \Omega \vdash I : \text{void}(\alpha)$.

Proof. By induction on program flattening on instructions. Consider a method call $I = \tau \mathbf{x} := \mathbf{e.m}(\mathbf{e}_1, \dots, \mathbf{e}_n)$; such that $\Gamma, \Omega \vdash I : \text{void}(\alpha)$ and $\Gamma = (s, \Delta)$. This means that $\Gamma, \Omega \vdash_{\gamma_i} \mathbf{e}_i : \tau_i(\alpha_i)$, $\Gamma, \Omega \vdash_{\gamma} \mathbf{x} : \tau(\alpha)$ and $\Gamma, \Omega \vdash_{\gamma'} \mathbf{e} : \mathcal{C}(\beta)$ hold, for some $\gamma_i, \gamma, \gamma', \tau_i, \alpha_i, \tau, \alpha, \mathcal{C}$ and β (see Rule (C) of Figure 5). Applying the rules of Figure 3, the flattening of I is of the shape $\underline{J} [\tau] \mathbf{x} = \mathbf{x}_{n+1}.\mathbf{m}'(\mathbf{x}_1, \dots, \mathbf{x}_n)$; with $J = \tau_1 \mathbf{x}_1 := \mathbf{e}_1; \dots \tau_n \mathbf{x}_n := \mathbf{e}_n; \tau_{n+1} \mathbf{x}_{n+1} := \mathbf{e}$. Let Γ' for the environment that is equal to Γ but on the method signature s where:

$$\Gamma'(\mathbf{y}) = \begin{cases} \tau(\alpha) & \text{if } \mathbf{y} = \mathbf{x} \\ \tau_i(\alpha_i) & \text{if } \mathbf{y} \in \{\mathbf{x}_1, \dots, \mathbf{x}_n\} \\ \mathcal{C}'(\beta) & \text{if } \mathbf{y} = \mathbf{x}_{n+1} \\ \Gamma(\mathbf{y}) & \text{otherwise} \end{cases}$$

We have $\Gamma', \Omega \vdash J [\tau] \mathbf{x} = \mathbf{x}_{n+1}.\mathbf{m}'(\mathbf{x}_1, \dots, \mathbf{x}_n) : \text{void}(\alpha)$ and $\Gamma', \Omega \vdash J : \text{void}(\alpha)$ (sub-typing might be used). By induction hypothesis, there is a contextual typing environment Γ'' such that $\Gamma'', \Omega \vdash \underline{J} : \text{void}(\alpha)$ and $\forall \mathbf{x} \in I, \Gamma''(\mathbf{x}) = \Gamma'(\mathbf{x}) = \Gamma(\mathbf{x})$ and, consequently, $\Gamma'' \vdash_{\gamma} \underline{I} : \text{void}(\alpha)$. All the other cases are treated similarly.

The converse is straightforward as Γ and Γ' match on the variables that they “share” in common. \square

4.9. Subject reduction

We introduce an intermediate lemma stating that an instruction obtained through the evaluation of the computational instruction of a well-typed program is also well-typed. For this, we first need to extend the type system so that it will be defined on any meta-instruction:

$$\frac{}{\Gamma, \Omega \vdash \text{pop}; : \text{void}(\alpha)} \text{(Pop)} \quad \frac{}{\Gamma, \Omega \vdash \text{push}(s_{\mathcal{H}}); : \text{void}(\mathbf{0})} \text{(Push)}$$

Figure 8: Extra typing rules for Push and Pop instructions

Notice that this definition is quite natural as `push` makes the memory increase and so is of tier $\mathbf{0}$ while `pop` makes the memory decrease and so can be of any tier. One possibility would have been to include these typing rules directly in the initial type system. We did the current choice as these meta-instructions

are only obtained during computation whereas the type inference is supposed to be performed statically on the flattened program, that is an AOO program, independently of its execution.

Lemma 2 (Subject reduction). *Let Γ be a contextual typing environment, Ω be an operator typing environment and P be an AOO program of executable `Exe{void main(){Init Comp}}` such that P is well-typed with respect to Γ and Ω . If $(\mathcal{I}, \underline{\text{Comp}}) \rightarrow^* (\mathcal{C}, \text{MI})$ then $\Gamma, \Omega \vdash \text{MI} : \text{void}(\mathbf{1})$.*

Proof. By Proposition 1, $\underline{\text{Comp}}$ can be typed. We proceed by induction on the reduction length n of \rightarrow ; \rightarrow^n being a notation for a length n reduction:

- If $n = 0$ then $\text{MI} = \underline{\text{Comp}}$ and since the program is well-typed, we have $\Gamma, \Omega \vdash \underline{\text{Comp}} : \text{void}(\mathbf{1})$.
- Now consider a reduction of length $n+1$. It can be written as $(\mathcal{I}, \underline{\text{Comp}}) \rightarrow^n (\mathcal{C}, \text{MI}) \rightarrow (\mathcal{C}', \text{MI}')$. By induction hypothesis $\Gamma, \Omega \vdash \text{MI} : \text{void}(\mathbf{1})$. Moreover, the last rule can be:
 - the evaluation of an assignment (Rules (1-7) of Figure 4) or the evaluation of a push or pop instructions (Rules (9) and (10) of Figure 4). In all these cases, $\text{MI} = \text{MI}_1 \text{MI}'$ and $(\mathcal{C}, \text{MI}) \rightarrow (\mathcal{C}', \text{MI}')$. By Rule (*Seq*) of Figure 6, $\Gamma, \Omega \vdash \text{MI}' : \text{void}(\alpha)$, for some α and, consequently, $\Gamma, \Omega \vdash \text{MI}' : \text{void}(\mathbf{1})$ using Rule (*ISub*) of Figure 6.
 - the evaluation of a method call (Rule (8) of Figure 4). In this case, $(\mathcal{C}, \text{MI}) = (\mathcal{C}, [\text{x}:=]\text{z.m}(\overline{\text{y}}); \text{MI}') \rightarrow (\mathcal{C}', \text{push}(s_{\mathcal{H}}); \text{MI}'' [\text{x}:=\text{z}']; \text{pop}; \text{MI})$, provided that MI'' is the flattened body of method m . By Rule (*Seq*) of Figure 6, $\Gamma, \Omega \vdash \text{MI}' : \text{void}(\alpha)$, for some α . By the extra rules of Figure 8, $\Gamma, \Omega \vdash \text{push}(s_{\mathcal{H}}); : \text{void}(\mathbf{0})$ and $\Gamma, \Omega \vdash \text{pop}; : \text{void}(\mathbf{0})$. Moreover by Rules (*Body*) of Figure 6 and (*C*) of Figure 5, the flattened body MI'' can be typed by $\Gamma, \Omega \vdash \text{MI}'' : \text{void}(\beta)$, for some β (the method output tier). We let the reader check that $\text{x}:=\text{z}'$; can also be typed using the same reasoning in the case where the method returns something. Finally, using several times Rule (*Seq*) and one time Rule (*ISub*) of Figure 6, we obtain that $\Gamma, \Omega \vdash \text{push}(s_{\mathcal{H}}); \text{MI}'' [\text{x}:=\text{z}']; \text{pop}; \text{MI} : \text{void}(\mathbf{1})$.
 - the evaluation of a while loop (Rules (11) and (12) of Figure 4). In this case $\text{MI} = \text{while}(\text{x})\{\text{MI}_1\} \text{MI}_2$ and either x is evaluated to false, in which case we take $\text{MI}' = \text{MI}_2$ or x evaluates to true and $\text{MI}' = \text{MI}_2 \text{while}(\text{x})\{\text{MI}_1\}$. In both cases, $\Gamma, \Omega \vdash \text{MI}_2 : \text{void}(\alpha)$, some α , and $\Gamma, \Omega \vdash \text{while}(\text{x})\{\text{MI}_1\} : \text{void}(\mathbf{1})$ can be derived by Rules (*Seq*) and (*Wh*) of Figure 6. Consequently, we can derive $\Gamma, \Omega \vdash \text{MI}' : \text{void}(\mathbf{1})$ by either using Rule (*ISub*) (for false), if needed, or just by using Rule (*Seq*) (for true).
 - the evaluation of a `if` (Rule (13) of Figure 4) can be done in a similar manner.

and so the result. \square

Remark 1. *Subject reduction is presented in a weak form as we do not consider the tier 0 case. It also holds but is of no interest in this particular case as, by monotonicity of the typing rule, every sub-instruction will be of tier 0 and there will be no loop and no recursive call. This is the reason why the computational instruction is typed using tier 1 in the definition of well-typedness.*

5. Safe recursion

5.1. Recursive methods

Given two methods of signatures s and s' and names m and m' , define the relation \sqsubset on method signatures by $s \sqsubset s'$ if m' is called in in the body of m . This relation is extended to inheritance by considering that overriding methods are called by the overridden method. Let \sqsubset^+ be its transitive closure. A method of signature s is *recursive* if $s \sqsubset^+ s$ holds. Given two method signatures s and s' , $s \equiv s'$ holds if both $s \sqsubset^+ s'$ and $s' \sqsubset^+ s$ hold. Given a signature s , the class $[s]$ is defined as usual by $[s] = \{s' \mid s' \equiv s\}$. Finally, we write $s \sqsubseteq^+ s'$ if $s \sqsubset^+ s'$ holds but not $s' \sqsubset^+ s$.

Notice that the extension of \sqsubset to method override is rough as we consider overriding methods to be called by the overridden method though it is clearly not always the case. However it is put in order to make the set $[s]$ computable for any method signature s . Indeed in a method call of the shape $x.m()$, provided that x is a declared variable of type C and that $\text{void } m()$ is a method declared in C and overridden in some subclass D , the evaluation may lead dynamically to either a call to $\text{void } m^C()$ or to a call to $\text{void } m^D()$ depending on the instance that will be assigned to x during the program evaluation. This choice is highly undecidable as it depends on program semantics. Here we will consider that $\text{void } m^C() \sqsubset \text{void } m^D()$. Consequently, if the call $x.m()$ is performed in the body of a method of signature s then $s \sqsubset \text{void } m^C() \sqsubset \text{void } m^D()$ and, consequently, $s \sqsubset^+ \text{void } m^D()$, *i.e.* a call to the method of signature s may lead to a call to the method of signature $\text{void } m^D()$.

Lemma 3. *Given an AOO program, for any method signature s , the set $[s]$ is computable in polynomial time in the size of the program.*

Proof. It just suffices to look at the program syntax to generate linearly the relation \sqsubset and then to compute its transitive closure \sqsubset^+ . \square

5.2. Level

The notion of level of a meta-instruction is introduced to compute an upper bound on the number of nested recursive calls for a method call evaluation.

Definition 4 (Level). *The level λ of a method signature is defined by:*

- $\lambda(s) = \max\{\lambda(s') \mid s \sqsubset s'\}$ if $s \notin [s]$ (*i.e.* the method is not recursive),
- $\lambda(s) = 1 + \max\{\lambda(s') \mid s \sqsubseteq^+ s'\}$ otherwise,

setting $\max(\emptyset) = 0$.

Let $\lambda(P)$ be the maximal level of a method in a program P (or \underline{P}). By abuse of notation, we will write $\lambda(\mathbf{m})$ and, respectively, λ when the signature of the method \mathbf{m} and the program P , respectively, are clear from the context.

Example 12. Consider the methods of Example 8:

- $\lambda(\text{getQueue}) = \lambda(\text{getValue}) = \lambda(\text{setQueue}) = 0$ as these methods do not call any other methods in their body and, consequently, are not recursive,
- $\lambda(\text{isEqual}) = 0$ as `isEqual` call the methods `getQueue` and `getValue` but the converse does not hold. Consequently, $\text{BList isEqual}^{\text{BList}}(\text{BList}) \not\equiv^+$ $\text{BList getQueue}^{\text{BList}}()$ and $[\text{BList isEqual}^{\text{BList}}(\text{BList})] = \emptyset$,
- $\lambda(\text{decrement}) = 1 + \max\{\lambda(\mathbf{s}') \mid \text{void decrement}^{\text{BList}}() \not\equiv^+ \mathbf{s}'\} = 1 + \max(\emptyset) = 1$,
- $\lambda(\text{length}) = 1$, for the same reason.

5.3. Intricacy

The notion of intricacy corresponds to the number of nested `while` loops in a meta-instruction and will be used to compute the requested upper bounds.

Definition 5 (Intricacy). Let the intricacy ν be partial function from meta-instructions to integers defined as follows:

- $\nu([\tau]\mathbf{x}:=\mathbf{me};) = 0$ if \mathbf{me} is not a method call,
- $\nu([\tau]\mathbf{x}:=\mathbf{y.m}(\dots);) = \max_{\mathbf{D} \leq \mathbf{C}^*}(\nu(\text{MI}_{\mathbf{D}}))$
provided that \mathbf{m} is of the shape $\tau \mathbf{m}(\dots)\{\text{MI}_{\mathbf{D}} [\text{return } \mathbf{z};]\}$ in a class $\mathbf{D} \leq \mathbf{C}^*$, where \mathbf{y} is of type \mathbf{C} and \mathbf{C}^* is the least super-class of \mathbf{C} where \mathbf{m} is defined.
- $\nu(\text{MI MI}') = \max(\nu(\text{MI}), \nu(\text{MI}'))$
- $\nu(\text{if}(\mathbf{x})\{\text{MI}\}\text{else}\{\text{MI}'\}) = \max(\nu(\text{MI}), \nu(\text{MI}'))$
- $\nu(\text{while}(\mathbf{x})\{\text{MI}\}) = 1 + \nu(\text{MI})$

Moreover, let $\nu(P)$ be the maximal intricacy of a meta-instruction within the flattened AOO program \underline{P} . By abuse of notation, we will write ν when the program P is clear from the context.

Observe that for any instruction \mathbf{I} , the intricacy of its flattening $\nu(\underline{\mathbf{I}})$ is well-defined as there is no `push` or `pop` operations occurring in it. Moreover, in the simple case where there is no inheritance then the intricacy of a method call $\nu([\tau]\mathbf{x}:=\mathbf{y.m}(\dots);)$, for some method of the shape $\tau \mathbf{m}(\dots)\{\text{MI} [\text{return } \mathbf{z};]\}$ is just equal to $\nu(\text{MI})$ (as $\mathbf{C}^* = \mathbf{C}$ and the only \mathbf{D} such that $\mathbf{D} \leq \mathbf{C}$ is \mathbf{C} itself).

Example 13. Consider the following meta-instruction MI :

```

while(x){
  while(y){
    b := l.isEqual(o);
  }
}

```

$\nu(\text{MI}) = 2 + \nu(\text{MI}')$, if MI' is the flattened body of the method `isEqual`. We let the reader check that $\nu(\text{MI}') = 1$ (there is one while inside). Consequently, $\nu(\text{MI}) = 3$.

5.4. Safety

Now we put some aside restrictions on recursive methods to ensure that their computations remain polynomially bounded.

Definition 6 (Safety). *A well-typed program with respect to a typing environment Δ and operator typing environment Ω is safe if for each recursive method $\tau \text{ m}(\overline{\tau \mathbf{x}})\{\mathbf{I} \text{ [return } \mathbf{x};]\}$:*

1. *there is exactly one call to some $\mathbf{m}' \in [\mathbf{m}]$ in the instruction \mathbf{I} ,*
2. *there is no while loop inside \mathbf{I} , i.e. $\nu(\mathbf{I}) = 0$,*
3. *and only judgments of the shape $(s, \Delta), \Omega \vdash_{\mathbf{1}} \tau \text{ m}^c(\overline{\tau \mathbf{x}}) : \mathbf{C}(\mathbf{1}) \times \langle \overline{\tau(\mathbf{1})} \rangle \rightarrow \tau(\alpha)$ can be derived using Rules (Body) and (OR).*

Item 1 ensures that recursive methods will be restricted to have only one recursive call performed during the evaluation of their body. It will prevent exponential accumulation through mutual recursion. A counter-example is a program computing the Fibonacci sequence with a mutually recursive call of the shape $\mathbf{m}(n-1) + \mathbf{m}(n-2)$, for $n > 2$. Item 2 is here to simplify the complexity analysis. It is not that restrictive in the sense that it enforces the programmer to make a choice between a functional programming style using pure recursive calls or an imperative one using while loops. Item 3 is very important. It enforces recursive methods to have tier **1** inputs in order to prevent a control flow depending on tier **0** variables during the recursive calls evaluation and to have an output whose use is restricted to tier **1** instructions (this is the purpose of the $\vdash_{\mathbf{1}}$ use) in order to prevent the recursive method call to be controlled by tier **0** instructions. The method output tier α is not restricted.

Example 14. *A well-typed program whose computational instruction uses the methods `getQueue`, `getValue`, `setQueue`, `length` and `isEqual` of Example 8 will be safe. Indeed the only recursive method is `length`. As illustrated in Example 8, it can be typed by $\text{BList}(\mathbf{1}) \rightarrow \text{int}(\mathbf{0})$, it does not contain any while loop and has only one recursive call in its body.*

*A program whose computational instruction uses the method `decrement` will not be safe as this method can only be given the type $\text{BList}(\mathbf{0}) \rightarrow \text{void}(\mathbf{0})$. Though it entails a lack of expressivity, there is no straightforward reason to reject this code. However we will see clearly in the next subsection that we do not want tier **0** data to control a while or a recursion while we do not want tier **1** data to be altered. Changing the type system by allowing `decrement` to apply to tier **1** objects would allow codes like:*

```

while(!o.isEqual(l)){
  o.decrement();
}

```

where l is a `BList` of n booleans equal to `false`. Clearly, such a loop can be executed exponentially in the input size (the size of the lists). This is highly undesirable.

An important point to mention is that safety allows an easy way to perform expression subtyping for objects through the use of cloning. In other words, it is possible to bring a reference type expression from tier `1` to tier `0` based on the premise that it has been cloned in memory (this was already true for primitive data by Rule *(Ass)*). Thus a form of subtyping that does not break the non-interference properties is admissible as illustrated by the following example:

Example 15. Let us add a clone method to the to the class `BList`. This clone method will output a copy of the current object that, as it is constructed, will be of tier `0`.

```

BList clone(){
  BList res0 := null;
  int v0 := value1;
  if(queue1 == null){
    res0 := new BList(v0,null)0;
  }
  else{
    res0 := new BList(v0,queue.clone()0);
  }
  return res;
}

```

This is typable by $\text{BList}(1) \rightarrow \text{BList}(0)$. Moreover, the method is safe, as there is only one recursive call.

Lemma 4. A program P is safe iff \underline{P} is safe.

Proof. By Proposition 1, P is well-typed iff \underline{P} is well-typed. The flattening transformation has no effect on method calls, while loops and method signatures. Consequently, P satisfies Items 1, 2 and 3 iff so does \underline{P} . \square

Lemma 5. Given a well-typed program P with respect to some typing derivation Π , the safety of P can be checked in polynomial time in $|P|$.

Proof. By Lemma 3, computing the sets of “equivalent” recursive methods can be done in polynomial time in $|P|$. Once, this is done, checking that there is no while loops and only one recursive call inside can be done quadratically in the program size. Finally, checking for Item 3 can be done linearly in the size of the typing derivation Π of P . However such a derivation is quadratic in the size of the program as all typing rules apart from *(ISub)* and *(Pol)* correspond to some program construct. Just notice that Rule *(ISub)* can be applied only once per instruction while Rule *(Pol)* can be applied at most k times per expression,

provided that k is an upper bound on the number of nested inheritance (and k is bounded by $|P|$). \square

Corollary 2. *Given a well-typed program P with respect to some typing derivation Π , the safety of \underline{P} can be checked in polynomial time in $|P|$.*

Proof. By Lemma 5, the safety of P can be checked in polynomial time in $|P|$ and so the result, by Lemma 4. \square

5.5. General safety

The safety criterion is sometimes very restrictive from an expressivity point of view because of Item 1. This Item is restrictive but simple: the interest is to have a decidable criterion for safety while keeping a completeness result for polynomial time as we shall see shortly. However it can be generalized to a semantics (thus undecidable criterion) that we call general safety in order to increase program expressivity.

Definition 7 (General safety). *A well-typed program with respect to a typing environment Δ and operator typing environment Ω is generally safe if for each recursive method $\tau \mathbf{m}(\overline{\tau \mathbf{x}})\{\mathbf{I} \text{ [return } \mathbf{x};]\}$:*

1. *the following condition is satisfied:*
 - (a) *either there is at most one call to some $\mathbf{m}' \in [\mathbf{m}]$ in the evaluation of \mathbf{I} ,*
 - (b) *or all recursive calls are performed on a distinct field of the current object. Such fields being used at most once.*
2. *there is no while loop inside \mathbf{I} , i.e. $\nu(\mathbf{I}) = 0$,*
3. *and only judgments of the shape $(s, \Delta), \Omega \vdash_{\mathbf{1}} \tau \mathbf{m}^c(\overline{\tau \mathbf{x}}) : \mathbf{C}(\mathbf{1}) \times \langle \overline{\tau(\mathbf{1})} \rangle \rightarrow \tau(\alpha)$ can be derived using Rules (Body) and (OR).*

Example 16. *General safety improves the expressivity of captured programs as illustrated by the following example:*

```
class Tree {
  int node;
  Tree left;
  Tree right;

  int value(BList b1) {
    int res1 := node1;
    if(b1 != null && right1 != null && left1 != null){
      if(b.getValue()){
        res := right.value(b.getQueue()1); : 1
      }else{
        res := left.value(b.getQueue()1); : 1
      }
    }else{;}
    return res;
  }
}
```

The method `value` returns the value of the node whose path is encoded by a boolean list in the method parameter `b` (the boolean constants `true` and `false` encoding the right and left sons, respectively). It can be typed by $\text{Tree}(\mathbf{1}) \times \text{BList}(\mathbf{1}) \rightarrow \text{int}(\mathbf{1})$. Consequently, it is generally safe (but not safe) as exactly one branch of the conditional (thus one recursive call) can be reached, thus satisfying Item 1a of Definition 7.

In the same manner, Item 1b allows us to implement the clone method on recursive structures:

```
Tree clone(){
  Tree res0 := null;
  int n0 := node1;
  if(left1 == null && right1 == null){
    //The case of non well-balanced trees is not treated.
    res0 := new Tree(n0,null,null)0;
  }
  else{
    res0 := new Tree(n0,left.clone()0, right.clone()0);
  }
  return res;
}
```

This is typable by $\text{Tree}(\mathbf{1}) \rightarrow \text{Tree}(\mathbf{0})$ thanks to Item 1b of Definition 7 as the two recursive calls are applied exactly once on distinct fields (`left` and `right`).

We can now compare safety and general safety:

Proposition 2. *If a program is safe then it is generally safe. The converse is not true.*

Proof. If there is only one recursive call in the body of any recursive method then, a fortiori, there is only one recursive call during the body evaluation. On the opposite in an instruction of the shape $I = \text{if}(e)\{m();\}\text{else}\{m();\}$ there is only one call of `m()` during the evaluation of `I` whereas `m` is called twice in `I`. \square

Proposition 3. *General safety is undecidable.*

Proof. Since Item 1a can be reduced to either a dead-code problem or a reachability problem that are known to be undecidable problems. Items 1b, 2 and 3 are still decidable in polynomial time. \square

Note that there are decidable classes of programs between safe programs and generally safe programs. For example, one may ask a program to have only one recursive call per reachable branch of a conditional in a method body. This requirement is clearly decidable in polynomial time.

Hence general safety can be seen as a generalization of the safe recursion on notation (SRN) scheme by Bellantoni and Cook [1]. Indeed a SRN function can be defined (and typed) by a method `f` in a class `C` by:

```

int f(int x, int y){
  int res = 0;
  if (x==0) {
    res = g(y);
  } else {
    if (x%2 == 0) {
      res = h0(f(x/2,y));
    } else {
      res = h1(f(x/2,y));
    }
  }
  return res;
}

```

This can be typed provided that $f : \mathbf{C}(\mathbf{1}) \times \mathbf{int}(\mathbf{1}) \times \mathbf{int}(\mathbf{1}) \rightarrow \mathbf{int}(\mathbf{0})$, $h_i : \mathbf{C}(\mathbf{1}) \times \mathbf{int}(\mathbf{0}) \rightarrow \mathbf{int}(\mathbf{0})$, $i \in \{0, 1\}$ and $g : \mathbf{C}(\mathbf{1}) \times \mathbf{int}(\mathbf{1}) \rightarrow \mathbf{int}(\mathbf{0})$. As f is recursive, its parameters have to be of tier $\mathbf{1}$. However its output can be of tier $\mathbf{0}$ provided that we can use only derivations of the shape $\Gamma, \Omega \vdash_{\mathbf{1}} \mathbf{int} f() : \mathbf{C}(\mathbf{1}) \times \mathbf{int}(\mathbf{1}) \times \mathbf{int}(\mathbf{1}) \rightarrow \mathbf{int}(\mathbf{0})$ and $\Gamma, \Omega \vdash_{\mathbf{1}} h_i : \mathbf{C}(\mathbf{1}) \times \mathbf{int}(\mathbf{0}) \rightarrow \mathbf{int}(\mathbf{0})$. If f output is of tier $\mathbf{0}$ (i.e. computes something) then h_i will not be able to recurse on it (as in SRN). Clearly, the above program fullfills the general safety criterion for some typing context Γ such that $\Gamma(x) = \Gamma(y) = \mathbf{int}(\mathbf{1})$ and $\Gamma(\mathbf{res}) = \mathbf{int}(\mathbf{0})$ as the recursive calls that are performed in the $\mathbf{res} = h_0(f(x/2, y))$; instruction can be given the type $\mathbf{void}(\mathbf{1})$ using Rules (*Ass*) and (*ISub*). The operators $\%2$ and $== 0$ can be given the type $\Omega(\%2) \ni \mathbf{int}(\mathbf{1}) \rightarrow \mathbf{int}(\mathbf{1})$ and $\Omega(== 0) \ni \mathbf{int}(\mathbf{1}) \rightarrow \mathbf{boolean}(\mathbf{1})$ as they are neutral. Finally, the method body is typable using twice the Rule (*If*) on tier $\mathbf{1}$ guard and instructions.

6. Type system non-interference properties

In this section, we demonstrate that classical non-interference results are obtained through the use of the considered type system. For that purpose, we introduce some intermediate lemmata. Notice that all the results presented in this section hold for compatible pairs only (see Section 3.8).

The confinement Lemma expresses the fact that no tier $\mathbf{1}$ variables are modified by a command of tier $\mathbf{0}$.

Lemma 6 (Confinement). *Let P be an AOO program of computational instruction \mathbf{Comp} , Γ be a contextual typing environment and Ω be an operator typing environment such that P is (generally) safe with respect to Γ and Ω . If $\Gamma, \Omega \vdash \mathbf{Comp} : \mathbf{void}(\mathbf{0})$, then every variable assigned to during the execution of $(\mathcal{I}, \mathbf{Comp})$ is of tier $\mathbf{0}$.*

Proof. By Proposition 1 and Lemma 4, we know that \mathbf{Comp} is safe with respect to environments Γ' and Ω such that $\Gamma', \Omega \vdash \mathbf{Comp} : \mathbf{void}(\mathbf{0})$. Now we prove by induction that for any MI such that $(\mathcal{I}, \mathbf{Comp}) \rightarrow^* (\mathcal{C}, \mathbf{MI})$, every variable assigned to in the evaluation of MI is of tier $\mathbf{0}$:

- if $\mathbf{MI} = \epsilon$ or $;$ then the result is straightforward.

- if $\text{MI} = [\tau] \text{ x} := \text{me};$. Assume x is of tier **1**. From rule (*Ass*), it means that $\alpha' = \mathbf{1}$, implying that $\alpha = \mathbf{1}$, then that $\text{MI} : \text{void}(\mathbf{1})$.
- $\text{MI} = \text{I}_1 \text{ I}_2$ (or $\text{if}(\text{x})\{\text{I}_1\}\text{else}\{\text{I}_2\}$) then both I_i are of tiered type $\text{void}(\mathbf{0})$ by Rule (*Seq*) (respectively (*If*)) and so the result by induction.
- $\text{MI} = \text{y.m}(\bar{\text{x}});$ then $\text{y.m}(\bar{\text{x}});$ is of tiered type $\text{void}(\mathbf{0})$ by Rule (*Ass*). Hence m cannot be a recursive method because of safety. More, m should be typed as $\Gamma', \Omega \vdash_{\gamma} \text{void m}^c(\bar{\tau}) : \mathcal{C}(\beta) \times \langle \bar{\tau}(\bar{\alpha}) \rangle \rightarrow \text{void}(\mathbf{0})$. Finally, by Rule (*Body*) the flattened body of m is also of tier $\text{void}(\mathbf{0})$. Consequently, by induction hypothesis, it does not contain assignments of tier **1** variables.

and so the result. \square

Lemma 7. *Let P be an AOO program of computational instruction Comp , Γ be a contextual typing environment and Ω be an operator typing environment such that P is (generally) safe with respect to Γ and Ω . For all MI such that $(\mathcal{I}, \text{Comp}) \rightarrow^* (\mathcal{C}, \text{MI})$ and $\Gamma, \Omega \vdash \text{MI} : \text{void}(\mathbf{0})$, there is no while loops and recursive calls evaluated during this execution.*

Proof. From Lemma 2, we know that all instructions reached from a tier **0** instruction are of tier **0**.

- By rule (*Wh*), if a while loop occurs, it is of tier **1**.
- By safety assumption, calls to recursive functions are of tier **1**. \square

We now establish a non-interference Theorem which states that if x is variable of tier **1** then the value stored in x is independent from variables of tier **0**. For this, we first need to define the suitable relation on memory configurations and meta-instructions:

Definition 8. *Let Γ be a contextual typing environment and Ω be an operator typing environment.*

- *The equivalence relation $\approx_{\Gamma, \Omega}$ on memory configurations is defined as follows:*
 $\mathcal{C} \approx_{\Gamma, \Omega} \mathcal{D}$ *iff the transitive closure of the pointer graph of \mathcal{C} corresponding to **1** variables with respect to Γ is equal to the one of \mathcal{D} and both stacks match everywhere except on tier **0** variables.*
- *The relation $\approx_{\Gamma, \Omega}$ is extended to commands as follows:*
 1. *If $\text{I} = \text{J}$ then $\text{I} \approx_{\Gamma, \Omega} \text{J}$*
 2. *If $\Gamma, \Omega \vdash \text{I} : \text{void}(\mathbf{0})$ and $\Gamma, \Omega \vdash \text{J} : \text{void}(\mathbf{0})$ then $\text{I} \approx_{\Gamma, \Omega} \text{J}$*
 3. *If $\text{I} \approx_{\Gamma, \Omega} \text{J}$ and $\text{K} \approx_{\Gamma, \Omega} \text{L}$ then $\text{I K} \approx_{\Gamma, \Omega} \text{J L}$*
- *Finally, it is extended to configurations as follows:*
If $\text{I} \approx_{\Gamma, \Omega} \text{J}$ and $\mathcal{C} \approx_{\Gamma, \Omega} \mathcal{D}$ then $(\mathcal{C}, \text{I}) \approx_{\Gamma, \Omega} (\mathcal{D}, \text{J})$

In other words, two commands that configurations are equivalent for $\approx_{\Gamma, \Omega}$ if their memory configurations restricted to tier **1** are the same and their commands of tier **1** are the same.

Theorem 1 (Non-interference). *Assume that Γ is a contextual typing environment and Ω is an operator typing environment such that MI and MJ are the computational instructions of two safe programs with respect to Γ and Ω having exactly the same classes. Assume also that $(\mathcal{C}, \text{MI}) \approx_{\Gamma, \Omega} (\mathcal{D}, \text{MJ})$. If $(\mathcal{C}, \text{MI}) \rightarrow (\mathcal{C}', \text{MI}')$ then there exist \mathcal{D}' and MJ' such that:*

- $(\mathcal{D}, \text{MJ}) \rightarrow^* (\mathcal{D}', \text{MJ}')$
- and $(\mathcal{C}', \text{MI}') \approx_{\Gamma, \Omega} (\mathcal{D}', \text{MJ}')$

Proof. We proceed by induction on MI .

- If $\text{MI} = \mathbf{x} := \mathbf{me};$. There are two cases to consider.
 - If $\Gamma, \Omega \vdash \text{MI} : \text{void}(\mathbf{0})$ then, by definition of $\approx_{\Gamma, \Omega}$, $\Gamma, \Omega \vdash \text{MJ} : \text{void}(\mathbf{0})$. Indeed, either $\text{MJ} = \text{MI}$, hence can be typed by tier **0**, or it is of tier **0** (MJ cannot be a sequence of tier **0** instructions). Hence Lemma 6 tells us that every variable assigned to in MJ is of tier **0** and there exists \mathcal{D}' such that $(\mathcal{D}, \text{MJ}) \rightarrow^* (\mathcal{D}', \epsilon)$ and $(\mathcal{C}', \epsilon) \approx_{\Gamma, \Omega} (\mathcal{D}', \epsilon)$. This holds as \mathcal{C}' and \mathcal{D}' match on tier **1** values.
 - If $\Gamma, \Omega \vdash \text{MI} : \text{void}(\mathbf{1})$ and cannot be given the type $\text{void}(\mathbf{0})$ (i.e. both \mathbf{x} and \mathbf{e} are of tier **1**) then, by definition of $\approx_{\Gamma, \Omega}$, $\text{MI} = \text{MJ}$. We let the reader check that the evaluation of \mathbf{me} leads to the same tier **1** value as it only depends on tier **1** variables (the only difficulty is for non recursive methods that might have tier **0** arguments but this is straightforward as it just consists in inlining the method body that is also safe with respect to Γ and Ω). Consequently, $\mathcal{C}' \approx_{\Gamma, \Omega} \mathcal{D}'$ as the change on tier **1**
- If $\text{MI} = \text{MI}_1 \text{MI}_2$. There are still two cases to consider:
 - Either MI is of tiered type $\text{void}(\mathbf{0})$ then so is MJ and the result is straightforward.
 - Or MI is of tiered type $\text{void}(\mathbf{1})$ and $\text{MJ} = \text{MJ}_1 \text{MJ}_2$ with $\text{MI}_i \approx_{\Gamma, \Omega} \text{MJ}_i$. By induction hypothesis if $(\mathcal{C}, \text{MI}_1) \rightarrow (\mathcal{C}', \text{MI}'_1)$ there exists $(\mathcal{D}', \text{MJ}'_1)$ such that $(\mathcal{D}, \text{MJ}_1) \rightarrow^* (\mathcal{D}', \text{MJ}'_1)$ and $(\mathcal{C}', \text{MI}'_1) \approx_{\Gamma, \Omega} (\mathcal{D}', \text{MJ}'_1)$. Consequently, $(\mathcal{C}, \text{MI}) \rightarrow (\mathcal{C}', \text{MI}'_1 \text{MI}_2)$, $(\mathcal{D}, \text{MJ}_1 \text{MJ}_2) \rightarrow^* (\mathcal{D}', \text{MJ}'_1 \text{MJ}_2)$ and $(\mathcal{C}', \text{MI}'_1 \text{MI}_2) \approx_{\Gamma, \Omega} (\mathcal{D}', \text{MJ}'_1 \text{MJ}_2)$, by definition of $\approx_{\Gamma, \Omega}$.

And so on, for all the other remaining cases. □

Given a configuration \mathcal{C} and a meta-instruction MI of a safe program with respect to contextual typing environment Γ and operator typing environment Ω , the *distinct tier 1 configuration sequence* $\xi_{\Gamma, \Omega}(\mathcal{C}, \text{MI})$ is defined by:

- If $(\mathcal{C}, \mathbf{MI}) \rightarrow (\mathcal{C}', \mathbf{MI}')$ then:

$$\xi_{\Gamma, \Omega}(\mathcal{C}, \mathbf{MI}) = \begin{cases} \xi_{\Gamma, \Omega}(\mathcal{C}', \mathbf{MI}') & \text{if } \mathcal{C} \approx_{\Gamma, \Omega} \mathcal{C}' \\ \mathcal{C}.\xi_{\Gamma, \Omega}(\mathcal{C}', \mathbf{MI}') & \text{otherwise} \end{cases}$$

- If $(\mathcal{C}, \mathbf{MI}) = (\mathcal{C}, \epsilon)$ then $\xi_{\Gamma, \Omega}(\mathcal{C}, \mathbf{MI}) = \mathcal{C}$.

Informally, $\xi_{\Gamma, \Omega}(\mathcal{C}, \mathbf{MI})$ is a record of the distinct tier **1** memory configurations encountered during the evaluation of $(\mathcal{C}, \mathbf{MI})$. Notice that this sequence may be infinite in the case of a non-terminating program. The relation $\approx_{\Gamma, \Omega}$ is extended to sequences by $\epsilon \approx_{\Gamma, \Omega} \epsilon$ (ϵ being the empty sequence) and $\mathcal{C}.\xi \approx_{\Gamma, \Omega} \mathcal{D}.\xi'$ iff both $\mathcal{C} \approx_{\Gamma, \Omega} \mathcal{D}$ and $\xi' \approx_{\Gamma, \Omega} \xi$ hold.

Now we can show another non-interference property à la Volpano et al. [3] stating that given a safe program, traces (distinct tier **1** configuration sequence) do not depend on tier **0** variables.

Lemma 8 (Trace non-interference). *Given a meta-instruction \mathbf{MI} of a safe program with respect to environments Γ and Ω , let \mathcal{C} and \mathcal{D} be two memory configurations, if $\mathcal{C} \approx_{\Gamma, \Omega} \mathcal{D}$ then $\xi_{\Gamma, \Omega}(\mathcal{C}, \mathbf{MI}) \approx_{\Gamma, \Omega} \xi_{\Gamma, \Omega}(\mathcal{D}, \mathbf{MI})$.*

Proof. By induction on the reduction \rightarrow and a case analysis on meta-instructions \mathbf{MI} :

- If $\mathbf{MI} = \mathbf{x} := \mathbf{me}$ then there are two cases to consider:
 - either $\Gamma, \Omega \vdash \mathbf{MI} : \mathbf{void}(\mathbf{0})$. In this case, by Confinement Lemma 6, \mathbf{x} is of tier **0** and thus if $(\mathcal{C}, \mathbf{MI}) \rightarrow (\mathcal{C}', \epsilon)$ and $(\mathcal{D}, \mathbf{MI}) \rightarrow (\mathcal{D}', \epsilon)$, we have $\mathcal{C}' \approx_{\Gamma, \Omega} \mathcal{C} \approx_{\Gamma, \Omega} \mathcal{D} \approx_{\Gamma, \Omega} \mathcal{D}'$.
 - or $\Gamma, \Omega \vdash \mathbf{MI} : \mathbf{void}(\mathbf{1})$ and not $\Gamma, \Omega \vdash \mathbf{MI} : \mathbf{void}(\mathbf{0})$. In this case, \mathbf{x} is a tier **1** variable. However as $\mathcal{C} \approx_{\Gamma, \Omega} \mathcal{D}$, \mathbf{me} evaluates to the same value or reference under both memory configuration. For example, in the case of a variable assignment (Rule (1) of Figure 4), we have $(\mathcal{C}, [\tau] \mathbf{x} := \mathbf{y};) \rightarrow (\mathcal{C}[\mathbf{x} \mapsto \mathcal{C}(\mathbf{y})], \epsilon)$ and $(\mathcal{D}, [\tau] \mathbf{x} := \mathbf{y};) \rightarrow (\mathcal{D}[\mathbf{x} \mapsto \mathcal{D}(\mathbf{y})], \epsilon)$. However, as $\mathcal{C} \approx_{\Gamma, \Omega} \mathcal{D}$ and \mathbf{y} is of tier **1** by Rule (*Ass*) of Figure 6, $\mathcal{C}(\mathbf{y}) = \mathcal{D}(\mathbf{y})$. Consequently $\mathcal{C}[\mathbf{x} \mapsto \mathcal{C}(\mathbf{y})] \approx_{\Gamma, \Omega} \mathcal{D}[\mathbf{x} \mapsto \mathcal{D}(\mathbf{y})]$. All the other cases of assignments (operator, methods) can be treated in the same manner, the constructor case being excluded as it enforces the output to be of tier **0**.
- If $\mathbf{MI} = \mathbf{while}(\mathbf{x})\{\mathbf{MI}'\}$ then, by Rule (*Wh*) of Figure 6, \mathbf{x} is enforced to be of tier **1**. Consequently, $\mathcal{C}(\mathbf{x}) = \mathcal{D}(\mathbf{x})$ and, consequently, $(\mathcal{C}, \mathbf{MI}) \rightarrow (\mathcal{C}, \mathbf{MI}')$ and $(\mathcal{D}, \mathbf{MI}) \rightarrow (\mathcal{D}, \mathbf{MI}')$, for some \mathbf{MI}' , independently of whether the guard evaluates to **true** or **false**. As a consequence, $\xi_{\Gamma, \Omega}(\mathcal{C}, \mathbf{MI}) = \xi_{\Gamma, \Omega}(\mathcal{C}, \mathbf{MI}) \approx_{\Gamma, \Omega} \xi_{\Gamma, \Omega}(\mathcal{D}, \mathbf{MI}) = \xi_{\Gamma, \Omega}(\mathcal{D}, \mathbf{MI}')$.

All the other cases for meta-instructions can be treated in the same manner and so the result. \square

This Lemma implies that tier **1** variables do not depend on tier **0** variables.

Using Lemma 8, if a safe program evaluation encounters twice the same meta-instruction under two configurations equal on tier **1** variables then the considered meta-instruction does not terminate on both configurations.

Corollary 3. *Given a memory configuration \mathcal{C} and a meta-instruction MI of a safe program with respect to environments Γ and Ω , if $(\mathcal{C}, \text{MI}) \rightarrow^+ (\mathcal{C}', \text{MI})$ and $\mathcal{C} \approx_{\Gamma, \Omega} \mathcal{C}'$, then the meta-instruction MI does not terminate on memory configuration \mathcal{C} .*

Proof. Assume that during the transition $(\mathcal{C}, \text{MI}) \rightarrow^+ (\mathcal{C}', \text{MI})$ there is a \mathcal{C}'' such that $\mathcal{C}'' \approx_{\Gamma, \Omega} \mathcal{C}$ does not hold, then the distinct tier **1** configuration sequence $\xi_{\Gamma, \Omega}(\mathcal{C}, \text{MI})$ contains this \mathcal{C}'' . From the construction of the sequence, we deduce that $\xi_{\Gamma, \Omega}(\mathcal{C}, \text{MI})$ is of the shape $\dots \mathcal{C}'' \dots \xi_{\Gamma, \Omega}(\mathcal{C}', \text{MI})$. However, by Lemma 8, $\xi_{\Gamma, \Omega}(\mathcal{C}, \text{MI}) \approx_{\Gamma, \Omega} \xi_{\Delta_1}(\mathcal{C}', \text{MI})$, hence it is infinite and the meta-instruction MI does not terminate on memory configuration \mathcal{C} .

Otherwise, we are in a state (\mathcal{C}, MI) from which the set of variables of tier **1** will never change and containing either a while loop or a recursive call. Consequently, there is some \mathcal{C}'' such that $(\mathcal{C}', \text{MI}) \rightarrow^+ (\mathcal{C}'', \text{MI})$ and so on. This means that the meta-instruction MI does not terminate on \mathcal{C} . \square

7. Polynomial time soundness

It is possible to bound the number of distinct configurations of tier **1** variables that can be met during the execution of a program, that is the number of different equivalence classes for the \approx relation on configurations.

Lemma 9. *Given a safe program with respect to environments Γ and Ω of computational instruction Comp on input \mathcal{I} , the number of equivalence classes for $\approx_{\Gamma, \Omega}$ on configurations is at most $|\mathcal{I}|^{n_1}$ where n_1 is the number of tier **1** variables in the computational instruction.*

Proof. First, let us note that the nodes and internal edges of tier **1** of the pointer graph do not change: new nodes created by constructors can only be of tier **0** from Rule *(Cons)*. Field assignments can only be of tier **0** according to Rule *(Ass)*. Moreover, Rule *(Self)* enforces all the field of an object to have tiers matching that of the current object. Consequently, reference type variables of tier **1** may only point to nodes of the initial pointer graph corresponding to input \mathcal{I} . The number of such nodes is bounded by $|\mathcal{I}|$ as $|\mathcal{I}|$ bounds the number of nodes in the initial pointer graph.

Second, we look at primitive type variables. By Definition 3 of operator typing environments, only the output of neutral operators (or of methods of return type of tier **1**) can be applied. Indeed, they are the only operators to have a tier **1** output and in an assignment of the shape $\mathbf{x} := \text{op}(\bar{\mathbf{y}})$, if op is of type $\langle \bar{\tau}(\mathbf{0}) \rangle \rightarrow \tau(\mathbf{0})$ then \mathbf{x} is enforced to be of tier **0** by Rule *(Ass)* of Figure 6. Neutral operators are operators whose output is either a value of a constant domain (`boolean`, `char`, ...) and, hence, has a constant number of distinct

values, or whose output is a positive integer value smaller than one of its input (also tier **1** values by Definition 3). Consequently, they can have a number of distinct values in $O(|\mathcal{I}|)$, as, by definition, $|\mathcal{I}|$ bounds the initial primitive values stored in the initial pointer mapping.

To conclude, let the number of tier **1** variables be n_1 , the number of distinct possible configurations is $|\mathcal{I}|^{n_1}$. \square

Now we can show the soundness: a safe and terminating program terminates in polynomial time.

Theorem 2 (Polynomial time soundness). *If an AOO program of computational instruction Comp is safe with respect to environments Γ and Ω and terminates on input \mathcal{I} :*

$$(\mathcal{I}, \underline{\text{Comp}}) \rightarrow^k (C', \epsilon)$$

then:

$$k = O(|\mathcal{I}|^{n_1(\nu+\lambda)}).$$

Proof. For simplicity, each recursive call can be simulated through derecursion by a while loop instruction of intricacy λ (keeping the number of reduction steps preserved relatively to some multiplicative and additive constants). Indeed recursive calls do not contain while loops and have only one method call in their body. Consequently, the maximum intricacy of an equivalent program with no recursive call is $\lambda + \nu$. Now we prove the result by induction on the intricacy ν of the transformed program:

- if $\nu = 0$ then the program has no while loops. Consequently, $k = O(1) = O(|\mathcal{I}|^{n_1 \times 0})$
- if $\nu = k + 1$ then, by definition of intricacy, the program contains at least one while loop. Let $\text{while}(x)\{\text{MI}\}$ be the first outermost while loop of the program. We have that $\nu(\text{MI}) = k$. By Induction hypothesis, MI can be evaluated in time $O(|\mathcal{I}|^{n_1 k})$, that is $O(|\mathcal{I}|^{n_1(\nu-1)})$. By Lemma 9, the tier **1** variable x can take at most $O(|\mathcal{I}|^{n_1})$ distinct values. Consequently, by termination assumption, the evaluation of $\text{while}(x)\{\text{MI}\}$ will take at most $O(|\mathcal{I}|^{n_1} \times |\mathcal{I}|^{n_1(\nu-1)})$ steps, that is $O(|\mathcal{I}|^{n_1 \nu})$ steps. Indeed, by Corollary 3, we know that a terminating program cannot reach twice the same configuration on tier **1** variables for a fixed meta-instruction. Now it remains to see that the instruction in the context have only while loops of intricacy smaller than ν . So, by induction again, they can be evaluated in time $O(|\mathcal{I}|^{n_1 \nu})$. Finally the total time is the sum so in time $O(|\mathcal{I}|^{n_1 \nu})$. \square

The same kind of results can be obtained for generally safe programs:

Proposition 4. *If an AOO program of computational instruction Comp is generally safe with respect to environments Γ and Ω and terminates on input \mathcal{I} :*

$$(\mathcal{I}, \underline{\text{Comp}}) \rightarrow^k (C', \epsilon)$$

then:

$$k = O(|\mathcal{I}|^{n_1(\nu+\lambda)}).$$

As a side effect, we obtain polynomial upper bounds on both the stack size and the heap size of safe terminating programs:

Theorem 3 (Heap and stack size upper bounds). *If an AOO program of computational instruction Comp is safe with respect to environments Γ and Ω and terminates on input \mathcal{I} then for each memory configuration $\mathcal{C} = \langle \mathcal{H}, \mathcal{S}_{\mathcal{H}} \rangle$ and meta-instruction MI such that $(\mathcal{I}, \underline{\text{Comp}}) \rightarrow^* (\mathcal{C}, \text{MI})$ we have:*

1. $|\mathcal{H}| = O(\max(|\mathcal{I}|, |\mathcal{I}|^{n_1(\nu+\lambda)}))$
2. $|\mathcal{S}_{\mathcal{H}}| = O(|\mathcal{I}|^{n_1(\nu+2\lambda)})$

Proof. (1) By Theorem 2, the number of reductions is in $O(|\mathcal{I}|^{n_1(\nu+\lambda)})$. The only instructions making the heap (pointer graph) increase are constructor calls. The number of such calls is thus bounded by $O(|\mathcal{I}|^{n_1(\nu+\lambda)})$ and consequently the heap size is bounded by the size of the original heap (that is in $O(|\mathcal{I}|)$) plus the size of the added nodes. Consequently, $|\mathcal{H}| = O(\max(|\mathcal{I}|, |\mathcal{I}|^{n_1(\nu+\lambda)}))$ as if $f = O(f')$ and $g = O(g')$ then $f + g = O(\max(f', g'))$.

(2) The number of stack frames added is bounded by $O(|\mathcal{I}|^{n_1\lambda})$ as, for some recursive method call of signature s , each level of recursion may add at most $O(|\mathcal{I}|^{n_1})$ stack frames corresponding to method signatures in the set $[s]$. Remember that a stack frame is just a signature of constant size 1 and a pointer mapping. The size of a pointer mapping is constant on boolean, character and reference type variables (it is equal to 1) and corresponds to the discrete value stored for numerical primitive type variables. As such values may only augment by a constant for each call to a positive operator and, as such calls may happen $O(|\mathcal{I}|^{n_1(\nu+\lambda)})$ times, we obtain that each stack frame has size bounded by $O(|\mathcal{I}|^{n_1(\nu+\lambda)})$. This is also due to the fact that no fresh variable is generated: consequently, the domain of pointer mapping is constantly bounded by the flattened program size, that is linear in the original program size by Corollary 1. Here we clearly understand the advantage to deal with flattened programs! All together, we obtain that $|\mathcal{S}_{\mathcal{H}}| = O(|\mathcal{I}|^{n_1\lambda} \times |\mathcal{I}|^{n_1(\nu+\lambda)})$, that is $O(|\mathcal{I}|^{n_1(\nu+2\lambda)})$. \square

Again the same results hold for generally safe programs:

Corollary 4. *If an AOO program of computational instruction Comp is generally safe with respect to environments Γ and Ω and terminates on input \mathcal{I} then for each memory configuration $\mathcal{C} = \langle \mathcal{H}, \mathcal{S}_{\mathcal{H}} \rangle$ and meta-instruction MI such that $(\mathcal{I}, \underline{\text{Comp}}) \rightarrow^* (\mathcal{C}, \text{MI})$ we have:*

1. $|\mathcal{H}| = O(\max(|\mathcal{I}|, |\mathcal{I}|^{n_1(\nu+\lambda)}))$
2. $|\mathcal{S}_{\mathcal{H}}| = O(|\mathcal{I}|^{n_1(\nu+2\lambda)})$

Example 17. *The AOO program of Example 7. This program is clearly terminating and safe (there is no recursive method call) with respect to the provided environments. Moreover intricacy ν is equal to 1 since there is no nested while loops in the method `getTail`. Moreover, its level is equal to 0 as there is no recursive call. Moreover there is one tier 1 variable $n_1 = 1$. Consequently, applying*

Theorem 2, we obtain that it terminates in $O(n^1)$, on some input of size n . Moreover, by Corollary 4, the heap size and stack size are in $O(\max(n, n^1)) = O(n)$ and $O(n^{1(1+2 \times 0)}) = O(n)$.

Example 18. Consider the below example representing cyclic data:

```

Ring {
  boolean data;
  Ring next;
  Ring prev;

  Ring(boolean d, Ring old) {
    data = d;
    if (old == null) {
      next = this;
      prev = this;
    } else {
      next = old.next;
      next.setPrev(this);
      prev = old.prev;
      prev.setNext(this);
    }
  }

  boolean getData() {return data;}
  Ring getNext() {return next;}
  Ring getPrev() {return prev;}
  void setPrev(Ring p) {prev = p;}
  void setNext(Ring n) {next = n;}
}

Exe {
  void main() {
    // Init
    Ring a = new Ring(true, null);
    Ring input = new Ring(true, a);
    //Comp: Search for a false in the input.
    copy1 = input1;
    while (copy1.getData() != false) {
      copy1 = copy1.getNext();
    }
  }
}

```

The program is safe and can be typed with respect to the following judgments:

- $\text{getData}() : \text{Ring}(\mathbf{1}) \rightarrow \text{boolean}(\mathbf{1})$
- $\text{getNext}() : \text{Ring}(\mathbf{1}) \rightarrow \text{Ring}(\mathbf{1})$

with respect to environments Γ and Ω such that $\Gamma(\text{copy}) = \Gamma(\text{input}) = \text{Ring}(\mathbf{1})$. Notice that methods $\text{setNext}(\text{Ring } n)$ and $\text{setPrev}(\text{Ring } p)$ are not required to

be typed with respect to tiers as they only appear in the initialization instruction and, consequently, their complexity is not under control (they are just supposed to build the input). It is obvious that if the main program halts, it will do so in time linear in the size of the input. But it can loop infinitely if the ring does not contain any **false**. We obtain a bound through the use of Theorem 2, that is $O(n^{n_1 \times 1}) = O(n^2)$. Notice that this bound can be ameliorated to $O(n)$ at the price of a non-uniform formula by noticing that only 1 tier **1** variable occurs in the while loop (see the remarks about declassification). Notice that this program could be adapted and typed in while loops of the shape:

```
while(copy.getData() != false && n>0){
  ...
  n--;
}
```

and this would be still typable.

Consequently, we can see that the presented methodology does not only apply to trivial data structures but can take benefit of any complex object structure.

8. Completeness

Another direct result is that this characterization is complete with respect to the class of functions computable in polynomial time as a direct consequence of Marion's result [5] since both language and type system can be viewed as an extension of the considered imperative language. This means that the type system has a good expressivity. We start to show that any polynomial can be computed by a safe and terminating program. Consider the following method of some class **C** computing addition:

```
add(int x, int y){
  while(x1>0){
    x1 := x1-1; :1
    y0 := y0+1; :0
  }
  return y0;
}
```

It can be typed by $\mathbb{C}(\beta) \times \mathbf{int}(\mathbf{1}) \times \mathbf{int}(\mathbf{0}) \rightarrow \mathbf{int}(\mathbf{0})$, for any tier β , under the typing environment Δ such that $\Delta(\mathbf{add}^{\mathbb{C}})(x) = \mathbf{int}(\mathbf{1})$ and $\Delta(\mathbf{add}^{\mathbb{C}})(y) = \mathbf{int}(\mathbf{0})$. Notice that **x** is enforced to be of tier **1**, by Rules (*Wh*) and (*Op*) as it appears in the guard of a while loop. The operators > 0 and -1 are neutral. Consequently, they can be given the tiered types $\mathbf{int}(\mathbf{1}) \rightarrow \mathbf{boolean}(\mathbf{1})$ and, $\mathbf{int}(\mathbf{0}) \rightarrow \mathbf{int}(\mathbf{0})$, respectively, in Rule (*Op*). The operator $+1$ is positive and its tiered type is enforced to be $\mathbf{int}(\mathbf{0}) \rightarrow \mathbf{int}(\mathbf{0})$ by Rule (*Op*).

Consider the below method encoding multiplication:

```
mult(int x, int y){
  int z0 := 0;
  while(x1>0){
```

```

    x1 := x1-1;
    int u1 := y1;
    while(u1>0){
        u1 := u1-1;
        z0 := z0+1;
    }
}
return z0;
}

```

It can be typed by $\mathcal{C}(\beta) \times \text{int}(\mathbf{1}) \times \text{int}(\mathbf{1}) \rightarrow \text{int}(\mathbf{0})$, for any tier β , under the typing environment Δ such that $\Delta(\text{mult}^{\mathcal{C}})(x) = \Delta(\text{mult}^{\mathcal{C}})(y) = \Delta(\text{mult}^{\mathcal{C}})(u) = \text{int}(\mathbf{1})$ and $\Delta(\text{mult}^{\mathcal{C}})(z) = \text{int}(\mathbf{0})$. Notice that x and u are enforced to be of tier $\mathbf{1}$, by Rules *(Wh)* and *(Op)*. Moreover y is enforced to be of tier $\mathbf{1}$, by Rule *(Ass)* applied to instruction `int u=y;`, u being of tier $\mathbf{1}$. Finally, z is enforced to be of tier $\mathbf{0}$, by Rule *(Op)*, as its stored value increases in the expression `z+1;`.

Theorem 4 (Completeness). *Each function computable in polynomial time by a Turing Machine can be computed by a safe and terminating program.*

Proof. We show that every polynomial time function over binary words, encoded using the class `BList`, can be computed by a safe and terminating program. Consider a Turing Machine TM , with one tape and one head, which computes within n^k steps for some constant k and where n is the input size. The tape of TM is represented by two variables x and y which contain respectively the reversed left side of the tape and the right side of the tape. States are encoded by integer constants and the current state is stored in the variable `state`. We assign to each of these three variables that hold a configuration of TM the tier $\mathbf{0}$. A one step transition is simulated by a finite cascade of if-commands of the form:

```

if (y.getHead()0){
    if (state0 == 80){
        state0 = 30;:  $\mathbf{0}$ 
        x0 = new BList(false, x0);:  $\mathbf{0}$ 
        y0 = y.getTail()0;:  $\mathbf{0}$ 
    }else{...:  $\mathbf{0}$ }
}

```

The above command expresses that if the current read symbol is `true` and the state is 8, then the next state is 3, the head moves to the right and the read symbol is replaced by `false`. The methods `getTail()` and `getHead()` can be given the types `BList($\mathbf{0}$) \rightarrow BList($\mathbf{0}$)` and `BList($\mathbf{0}$) \rightarrow boolean($\mathbf{0}$)`, respectively (see previous Example). Since each variable inside the above command is of tier $\mathbf{0}$, the tier of the if-command is also $\mathbf{0}$. As shown above, any polynomial can be computed by a safe and terminating program: we have already provided the programs for addition and multiplication and we let the reader check that it can be generalized to any polynomial. Thus the cascade of one step transitions can be included in an intrication of while loops computing the requested polynomial.

Note that this is possible as the cascade will be of tiered type `void(0)` and it can be typed by `void(1)` through the use of Rule (*ISub*). \square

Corollary 5. *Each function computable in polynomial time by a Turing Machine can be computed by a generally safe and terminating program.*

Proof. By Proposition 2. \square

9. Type inference

Now we show a decidability result for type inference in the case of safe programs.

Proposition 5 (Type inference). *Given an AOO program P , deciding if there exist a typing environment Δ and an operator typing environment Ω such that P is well-typed can be done in time polynomial in the size of the program.*

Proof. We work on the flattened program. Type inference can be reduced to a 2-SAT problem. We encode the tier of each field \mathbf{x} (respectively instruction MI) within the method \mathbf{m} of class \mathbf{C} by a boolean variable $x^{\mathbf{m}^c}$ (respectively $I^{\mathbf{m}^c}$) that will be true if the variable (instruction) is of tier **1**, false if it is of tier **0** in the context of \mathbf{m}^c . Then we generate boolean clauses with respect to the program code:

- An assignment $\mathbf{x} := \mathbf{y}$; corresponds to $(y^{\mathbf{m}^c} \wedge x^{\mathbf{m}^c})$
- A sequence $\text{MI}_1 \text{MI}_2$ corresponds to $(I_1^{\mathbf{m}^c} \vee I_2^{\mathbf{m}^c})$
- An `if(x){MI1}else{MI2}` corresponds to $(\neg I_1^{\mathbf{m}^c} \vee x^{\mathbf{m}^c}) \wedge (\neg I_2^{\mathbf{m}^c} \vee x^{\mathbf{m}^c})$
- A expression involving a neutral operator $\mathbf{x}_0 := \text{op}(\mathbf{x}_1, \dots, \mathbf{x}_n)$ corresponds to $\wedge_{i,j} (x_i^{\mathbf{m}^c} \vee \neg x_j^{\mathbf{m}^c})$
- and so on...

The number of generated clauses is polynomial in the size of the flattened program and also of the initial program by Corollary 1. The polynomiality comes from the fact that a method may be given distinct types on several calls (thus its body might be typed several times statically). Consequently, tiered types are inferred in polynomial time as 2-SAT problems can be solved in linear time. \square

Corollary 6. *Given an AOO program P , deciding if there exist a typing environment Δ and an operator typing environment Ω such that P is well-typed and safe can be done in time polynomial in the size of the program.*

Proof. Using Lemma 5. \square

Just remark that the above corollary becomes false for generally safe programs as a consequence of Proposition 3.

10. Extensions

In this section, we discuss several possible improvements of the presented methodology.

10.1. Control flow alteration

Constructs altering the control flow like break, return and continue can also be considered in this fragment. For example, a break statement has to be constrained to be of tiered type `void(1)` so that if such an instruction is to be executed, then we know that it does not depend on tier `0` expressions. More precisely, it can be typed by the rule:

$$\frac{}{\Gamma, \Omega \vdash \text{break}; : \text{void}(1)} \textit{(Break)}$$

This prevents the programmer from writing conditionals of the shape:

```
while(x1){...if(y0){break;}else{...}...}
```

that would break the non-interference result of Lemma 8. Such a conditional cannot be typed in such a way since `y` has to be of tier `1` by Rules *(If)* and *(Break)*. Notice that Theorem 1 remains valid since in a *terminating program*, an instruction containing break statements will have an execution time smaller than the same instruction where the break statements have been deleted. Using the same kind of typing rule, return statements can also be used in a more flexible manner by allowing the execution to leave the current subroutine anywhere in the method body. In previous Section, we have made the choice not to include break statement in the code as this make the definition of a formal semantics much more difficult: at any time, we need to keep in mind the innermost loop executed. The same remarks hold for the continue construct.

Another possible extension is to consider methods with no restriction on the return statement. For the simplicity of the type system, we did not present a flexible treatment of the return statements by allowing the execution to leave the current subroutine anywhere in the method body. But this can be achieved in the same manner. The difficulty here is that all the return constructs have to be of the same tiered type (this is a global check on the method body). Moreover, as usual, we need to enforce that in a conditional all reachable flows lead to some return construct.

10.2. Static methods, static variables and access modifiers

The exposed methodology can be extended without any problem to static methods since they can be considered as a particular case of methods. In a similar manner, static variables can be captured since they are global (and can be considered as variables of the executable class) and add no complexity to the program. We claim that the current analysis can handle all usual access modifiers. Indeed the presented work is based on the implicit assumption that all fields are `private` since there is no field access in the syntax. On the opposite, methods and classes are all `public`. Consequently, method access restriction only consists in restricting the class of analyzed programs.

10.3. Abstract classes and interfaces

Both abstract classes and interfaces can be analyzed by the presented framework. This is straightforward for interfaces since they do not add any complexity to the program. We claim that abstract classes can be analyzed since they are just a particular and simple case of inheritance.

10.4. Garbage Collecting

In the pointer graph semantics, dereferenced objects stay in memory forever. It does not entail the bound on the heap size, however, it is far from optimal from a memory usage point of view. However, it is easy to add naive garbage collecting to the system. Indeed, finding which objects in the graph are dereferenced is simply recursively deleting nodes whose indegree is zero (counting pointers in the indegree).

Two different strategies can be thought of to implement dynamically this idea: either use an algorithm similar to the mark-and-don't-sweep algorithm that will color the part of the graph that is referenced then delete the uncolored part; or noting that `assignment` and `pop` are the only instructions that may dereference a pointer, maintain for each node of the graph a counter for its indegree, update it at each instruction and delete the dereferenced nodes (and its children if they have no other parents) whenever it happens.

The first strategy has the main drawback that exploring the whole graph will block the execution for some time, especially as starting from each node in each pointer mapping of the pointer stack is needed.

The second strategy will be far more flexible and the real wiping of memory may be deferred if necessary (as the deleting of files in a Unix system for example). An assignment increments the indegree of the node assigned and decrements the indegree of the node previously assigned to the variable. An instantiation (`new`) adds arrows in the graph, hence increments the indegree of all the nodes associated to its parameter. `push` and `pop` respectively increment and decrement the indegree of the parameters and the current object of the method. Whenever a node's indegree reaches zero, it can be deleted. Finally, deleting a node of the graph decrements the indegree of the nodes it pointed to.

10.5. Alleviating the safety condition

Another way to alleviate the safety condition is to reuse the distinct recursion schemata provided in the ICC works of the function algebra. For example, over `Tree` structures, a recursive definition -written in a functional manner - of the shape:

```
f(t) = new Tree(f(t.getLeft()), f(t.getRight()));
```

should be accepted by the analysis as recursive calls work on partitioned data and, consequently, the number of recursive calls remains linear in the input size.

One sufficient condition to ensure that property is that:

- recursive calls of the method body are performed on distinct fields

- and these fields are never assigned to in the method body

The following depth first tree traversal algorithm satisfies this condition:

```
class Tree {
  int node;
  Tree left;
  Tree right;
  ...
  void visit() {
    println(node);
    if(left != null){
      left.visit();
    }else{;}
    if(right != null){
      right.visit();
    }else{;}
  }
}
```

on the assumption that the method `println(n)` behaves as expected. We let the reader check that the method `visit()` can be typed by $\text{Tree}(\mathbf{1}) \rightarrow \text{void}(\mathbf{1})$.

10.6. Declassification

In non-interference settings, declassification consists in lowering the confidentiality level of part of the data. For example, when verifying a password, the password database itself is at the highest level, knowing whether an input password matches should have the same level of confidentiality. As this is highly impractical, declassifying this partial information makes sense.

In this context, declassifying would mean retyping some tier $\mathbf{0}$ variables into $\mathbf{1}$. Such a flow is strictly forbidden by the type system, but it would make sense, for example, to compose treatments that are separately well-typed. As long as those treatments are in finite number, we keep the polynomial bound as bounded composition of polynomials remains polynomial.

Formally, we will say that programs of the form $\text{Exe}\{\text{void main}\{\text{Init } I_1 I_2 \dots I_n\}\}$ are well-typed iff for each $i \leq n$, $\text{Exe}\{\text{void main}\{\text{Init } I_1 \dots I_i\}\}$ is well-typed when we consider $\text{Init } I_1 \dots I_{i-1}$ to be the initialization instruction.

Example 19. *The following program:*

```
Exe {
  void main() {
    //Init
    int n := ...;
    BList b := null;
    while (n>0) {
      b := new BList(true, b);
      n := n-1;
    }
  }
}
```

```

}
//Comp1
z := 0;
while (y.getQueue()) {
  z := z+1
}
//Comp2
x := z;
BList c := null;
while (x>0) {
  c := new BList(false, c);
}
//Comp3
x := z;
while (x>0) {
  c := new BList(true, c);
}
}
}

```

cannot be typed without declassification as in **Comp1**, **z** needs to be of tier **0**, while in **Comp2** and **Comp3**, it needs to be of tier **1**.

Remark 2. Note that the proof of Theorem 4 could be improved by using declassification. Indeed, we could write a first computation instruction that creates a polynomial bound on the number of steps of the Turing Machine, and a second computation instruction that executes the simulation of each step while a counter is lower than the bound. In the first part, the bound needs to be of tier **0**, in the second of tier **1**, hence the use of declassification.

11. Related works

11.1. Related works on tier-based complexity analysis

The current work is inspired by three previous works:

- the seminal paper [5], initiating imperative programs type-based complexity analysis using secure information flow and providing a characterization of polynomial time computable functions,
- the paper [10], extending previous analysis to C processes with a fork/wait mechanism, which provides a characterization of polynomial space computable functions,
- and the paper [11], extending this methodology to a graph based language.

The current paper tries to pursue this objective but on a distinct paradigm: Object. It differs from the aforementioned works on the following points:

- first, it is an extension to the object-oriented paradigm (although imperative features can be dealt with). In particular, it characterizes the complexity of recursive and non-recursive method calls whereas previous works [5, 10, 11] were restricted to while loops and to non-object data type (words in [5, 10] and records in [11]),
- second, it studies program intensional properties (like heap and stack) whereas previous papers were focusing on the extensional part (characterizing function spaces). Consequently, it is closer to a programmer’s expectations,
- third, it provides explicit big O polynomial upper bounds while the two aforementioned studies were only certifying algorithms to compute a function belonging to some fixed complexity class.
- last, from an expressivity point of view, the presented results strictly extend the ones of [5], as the restriction of this paper to primitive data types is mainly the result of [5], while they are applied on a more concrete language than the ones in [11].

The current work is an extended version of [4]. The main distinctions are the following:

- In [4], only general safety is studied. Here we have presented a decidable (and thus restricted) safety condition.
- Contrarily to [4], the paper presents a formal semantics. The pros are a cleaner theoretical treatment. The cons are that we do not have constructs changing the flow like “break”. Such constructs can be handled by the tier-based type system. However, their use would increase drastically the complexity of the semantics as a program counter would be required.
- The requirement that a recursive method can only be called in a tier **1** instruction is formalized through the use of the \vdash_1 notation while it was only informally stated in [4].

11.2. Other related works on complexity

There are several related works on the complexity of imperative and object oriented languages. On imperative languages, the papers [12, 13, 14] study theoretically the heap-space complexity of core-languages using type systems based on a matrices calculus.

On OO programming languages, the papers [15, 16] control the heap-space consumption using type systems based on amortized complexity introduced in previous works on functional languages [17, 18, 19]. Though similar, the presented result differs on several points with this line of work. First, this analysis is not restricted to linear heap-space upper bounds. Second, it also applies to stack-space upper bounds. Last but not least, this language is not restricted to

the expressive power of method calls and includes a `while` statement, controlling the interlacing of such a purely imperative feature with functional features like recurrence being a very hard task from a complexity perspective.

Another interesting line of research is based on the analysis of heap-space and time consumption of Java bytecode [20, 21, 22, 23]. The results from [20, 21, 22] make use of abstract interpretations to infer efficiently symbolic upper bounds on resource consumption of Java programs. A constraint-based static analysis is used in [23] and focuses on certifying memory bounds for Java Card. The analysis can be seen as a complementary approach since we try to obtain practical upper bounds through a cleaner theoretically oriented treatment. Consequently, this approach allows the programmer to deal with this typing discipline on an abstract OO code very close to the original Java code without considering the corresponding Java bytecode. Moreover, this approach handles very elegantly while loops guarded by a variable of reference type whereas most of the aforementioned studies are based on invariants generation for primitive types only.

The concerns of this study are also related to the ones of [24, 25], that try to predict the minimum amount of heap space needed to run the program without exhausting the memory, but the methodology, the code analyzed (source vs compiled) and the goals differ.

A complex type-system that allows the programmer to verify linear properties on heap-space is presented in [26]. The presented result in contrast presents a very simple type system that however guarantees a polynomial bound.

In a similar vein, characterizing complexity classes below polynomial time is studied in [27, 28]. These works rely on a programming language called PURPLE combining imperative statements together with pointers on a fixed graph structure. Although not directly related, the presented type system was inspired by this work.

11.3. Related works on termination

The presented work is independent from termination analysis but the main result relies on such analysis. Indeed, the polynomial upper bounds on both the stack and the heap space consumption of a typed program provided by Theorem 3 only hold for a terminating computation. Consequently, this analysis can be combined with termination analysis in order to certify the upper bounds on any input. Possible candidates for the imperative fragment are *Size Change Termination* [29, 30], tools like Terminator [31] based on *Transition predicate abstraction* [32] or symbolic complexity bound generation based on abstract interpretations, see [33, 34] for example.

12. Conclusion

This work presents a simple but highly expressive type-system that is sound and complete with respect to the class of polynomial time computable functions, that can be checked in polynomial time and that provides explicit polynomial upper bounds on the heap size and stack size of an object oriented program

allowing (recursive) method calls. As the system is purely static, the bounds are not as tight as may be desirable. It would indeed be possible to refine the framework to obtain a better exponent at the price of a non-uniform formula (for example not considering all tier **1** variables but only those modified in each while loop or recursive method would reduce the computed complexity. See Example 18). OO features, such as abstract classes, interfaces and static fields and methods, were not considered here, but we claim that they can be treated by this analysis.

This analysis has several advantages:

- It provides a high-level alternative to usual complexity studies mainly based on the compiled bytecode. The analysis is high-level but the obtained bound are quite tight.
- It is decidable in polynomial time on the safety criterion.
- It merges both theoretical and applied results as we both obtain bounds on real programs and a sound and complete characterization of polynomial time computable functions.
- It is able to deal with the complexity of programs whose loops are guarded by object. This is not a feature of bytecode-based approaches that are restricted to primitive data and are in need of costly program transformation techniques and tools to comply with such kind of programs.
- It uses previous theoretical techniques (tiering, safe recursion on notation) for functional programs and function algebra and shows that they can be adapted elegantly (though technically) to the OO paradigm.
- It uses previous security techniques (non-interference, declassification) for imperative programs. The use is slightly different (even orthogonal) but the methodology is surprisingly very close.

We expect this paper to be a first step towards the use of tiers and non-interference for controlling the complexity of OO programs. The next steps are the design of a practical application and extensions to (linear or polynomial) space using threads.

Acknowledgments. The authors gratefully acknowledge the advises and comments from anonymous referees that contributed to improving this article.

- [1] S. Bellantoni, S. Cook, A new recursion-theoretic characterization of the poly-time functions, *Comput. Complex.* 2 (1992) 97–110.
- [2] D. Leivant, J.-Y. Marion, Lambda calculus characterizations of poly-time, *Fundam. Inform.* 19 (1/2) (1993) 167–184.
- [3] D. Volpano, C. Irvine, G. Smith, A sound type system for secure flow analysis, *J. Computer Security* 4 (2/3) (1996) 167–188.

- [4] E. Hainry, R. Péchoux, Objects in Polynomial Time, in: Programming Languages and Systems - 13th Asian Symposium, APLAS 2015, Vol. 9458 of Lecture Notes in Computer Science, 2015, pp. 387–404.
- [5] J.-Y. Marion, A type system for complexity flow analysis, in: 26th Annual IEEE Symposium on Logic in Computer Science, LICS 2011, 2011, pp. 123–132.
- [6] A. Igarashi, B. C. Pierce, P. Wadler, Featherweight java: a minimal core calculus for java and GJ, *ACM Trans. Program. Lang. Syst.* 23 (3) (2001) 396–450.
- [7] J.-Y. Girard, Light linear logic, *Inf. Comput.* 143 (2) (1998) 175–204.
- [8] S. A. Cook, C. Rackoff, Space lower bounds for maze threadability on restricted machines, *SIAM J. Comput.* 9 (3) (1980) 636–652.
- [9] N. Danner, J. S. Royer, Ramified structural recursion and corecursion, *CoRR* abs/1201.4567.
URL <http://arxiv.org/abs/1201.4567>
- [10] E. Hainry, J.-Y. Marion, R. Péchoux, Type-based complexity analysis for fork processes, in: FOSSACS, Vol. 7794 of Lecture Notes in Computer Science, 2013, pp. 305–320.
- [11] D. Leivant, J.-Y. Marion, Evolving graph-structures and their implicit computational complexity, in: Automata, Languages, and Programming - ICALP 2013, Vol. 7966 of Lecture Notes in Computer Science, 2013, pp. 349–360.
- [12] K.-H. Niggl, H. Wunderlich, Certifying polynomial time and linear/polynomial space for imperative programs, *SIAM J. Comput.* 35 (5) (2006) 1122–1147.
- [13] J.-Y. Moyén, Resource control graphs, *ACM Trans. Comput. Logic* 10 (4) (2009) 29:1–29:44.
- [14] N. D. Jones, L. Kristiansen, A flow calculus of *wp*-bounds for complexity analysis, *ACM Trans. Comput. Log.* 10 (4).
- [15] M. Hofmann, S. Jost, Type-based amortised heap-space analysis, in: ESOP, Vol. 3924 of Lecture Notes in Computer Science, 2006, pp. 22–37.
- [16] M. Hofmann, D. Rodriguez, Efficient type-checking for amortised heap-space analysis, in: CSL, Vol. 5771 of Lecture Notes in Computer Science, 2009, pp. 317–331.
- [17] M. Hofmann, S. Jost, Static prediction of heap space usage for first-order functional programs, in: Symposium on Principles of Programming Languages, POPL 2003, ACM, 2003, pp. 185–197.

- [18] S. Jost, K. Hammond, H.-W. Loidl, M. Hofmann, Static determination of quantitative resource usage for higher-order programs, in: Symposium on Principles of Programming Languages, POPL 2010, 2010, pp. 223–236.
- [19] L. Beringer, M. Hofmann, A. Momigliano, O. Shkaravska, Automatic certification of heap consumption, in: LPAR, Vol. 3452 of Lecture Notes in Computer Science, 2004, pp. 347–362.
- [20] E. Albert, P. Arenas, S. Genaim, G. Puebla, D. Zanardini, Costa: Design and implementation of a cost and termination analyzer for java bytecode, in: FMCO, Vol. 5382 of Lecture Notes in Computer Science, 2008, pp. 113–132.
- [21] E. Albert, P. Arenas, S. Genaim, G. Puebla, D. Zanardini, Cost analysis of object-oriented bytecode programs, *Theor. Comput. Sci.* 413 (1) (2012) 142–159.
- [22] R. Kersten, O. Shkaravska, B. van Gastel, M. Montenegro, M. C. J. D. van Eekelen, Making resource analysis practical for real-time java, in: Java Technologies for Real-time and Embedded Systems, JTRES '12, 2012, pp. 135–144.
- [23] D. Cachera, T. Jensen, D. Pichardie, G. Schneider, Certified memory usage analysis, in: FM 2005: Formal Methods, Vol. 3582 of Lecture Notes in Computer Science, 2005, pp. 91–106.
- [24] E. Albert, S. Genaim, M. Gómez-Zamalloa Gil, Live heap space analysis for languages with garbage collection, in: Proceedings of the 2009 international symposium on Memory management, ACM, 2009, pp. 129–138.
- [25] E. Albert, S. Genaim, M. Gómez-Zamalloa, Parametric inference of memory requirements for garbage collected languages, in: ACM Sigplan Notices, Vol. 45, ACM, 2010, pp. 121–130.
- [26] W. Chin, H. Nguyen, S. Qin, M. Rinard, Memory usage verification for OO programs, in: Static Analysis, SAS 2005, 2005, pp. 70–86.
- [27] M. Hofmann, U. Schöpp, Pointer programs and undirected reachability, in: 24th Annual IEEE Symposium on Logic in Computer Science, LICS 2009, IEEE Computer Society, 2009, pp. 133–142.
- [28] M. Hofmann, U. Schöpp, Pure pointer programs with iteration, *ACM Trans. Comput. Log.* 11 (4).
- [29] A. M. Ben-Amram, Size-change termination, monotonicity constraints and ranking functions, *Log. Meth. Comput. Sci.* 6 (3).
- [30] A. M. Ben-Amram, S. Genaim, A. N. Masud, On the termination of integer loops, in: Verification, Model Checking, and Abstract Interpretation, VMCAI 2012, Vol. 7148 of Lecture Notes in Computer Science, 2012, pp. 72–87.

- [31] B. Cook, A. Podelski, A. Rybalchenko, Terminator: Beyond safety, in: Computer Aided Verification, CAV 2006, Vol. 4144 of Lecture Notes in Computer Science, 2006, pp. 415–426.
- [32] A. Podelski, A. Rybalchenko, Transition predicate abstraction and fair termination, in: Symposium on Principles of Programming Languages, POPL 2005, ACM, 2005, pp. 132–144.
- [33] S. Gulwani, Speed: Symbolic complexity bound analysis, in: Computer Aided Verification, CAV 2009, Vol. 5643 of Lecture Notes in Computer Science, 2009, pp. 51–62.
- [34] S. Gulwani, K. K. Mehra, T. M. Chilimbi, Speed: precise and efficient static estimation of program computational complexity, in: Symposium on Principles of Programming Languages, POPL 2009, ACM, 2009, pp. 127–139.

Résumé

In this thesis, we explore the different results obtained by the implicit complexity community over the past thirty years. These results generally consist in characterizing a given complexity class on a fixed programming language using a static analysis technique (interpretation, tiering, type system, light logic, ...). After listing the main techniques and highlighting their similarities, we will first study the intensionality results obtained for some these techniques, i.e. results that make it possible to compare the expressive power of these techniques or to extend the number of captured programs. Then, we will study crossovers, extensions of these techniques to other programming paradigms. We will then focus our analysis on the results obtained by using these techniques in the context of infinite data types such as streams, infinite sequences, real numbers, and higher-order functions. Finally, we will conclude this thesis by presenting some of the most interesting open questions in the field, including extensions to probabilistic models or quantum computing.

Abstract

Dans ce mémoire, nous explorons les différents résultats obtenus par la communauté de la complexité implicite au cours de ces trente dernières années. Ces résultats permettent en général de caractériser une classe de complexité donnée sur un langage de programmation fixé en utilisant une technique d'analyse statique (interprétations, tiering, systèmes de types, logiques légères, ...). Après avoir listé les principales techniques et avoir mis en avant leurs similitudes, nous étudierons dans un premier temps, les résultats d'intensionnalité obtenus pour certaines de ces techniques, c'est-à-dire des résultats permettant de comparer le pouvoir d'expression de ces techniques ou permettant d'étendre le nombre de programmes capturés. Puis, nous étudierons des "crossovers", des extensions de ces techniques à d'autres paradigmes de programmation. Nous focaliserons ensuite notre analyse sur les résultats obtenus en utilisant ces techniques dans le cadre de domaines infinis tels que les streams, les suites infinies, les nombres réels et les fonctions d'ordre supérieur. Enfin, nous conclurons ce mémoire en exposant certaines des questions ouvertes les plus intéressantes du domaine, dont des extensions à des modèles probabilistes ou de l'informatique quantique.

