Type Checking Higher-Order Polymorphic Multi-Methods

François Bourdoncle
Centre de Mathématiques Appliquées,
École des Mines de Paris
Francois.Bourdoncle@ensmp.fr

Stephan Merz
Institut für Informatik,
Universität München
merz@informatik.uni-muenchen.de

Abstract

We present a new predicative and decidable type system, called $\mathrm{ML}_{\leq}$, suitable for languages that integrate functional programming and parametric polymorphism in the tradition of ML [21, 28], and class-based object-oriented programming and higher-order multi-methods in the tradition of CLOS [12]. Instead of using extensible records as a foundation for object-oriented extensions of functional languages, we propose to reinterpret ML datatype declarations as abstract and concrete class declarations, and to replace pattern matching on run-time values by dynamic dispatch on run-time types. $\mathrm{ML}_{\leq}$ is based on universally quantified polymorphic constrained types. Constraints are conjunctions of inequalities between monotypes built from type constructors organized into extensible and partially ordered classes. We give type checking rules for a small, explicitly typed functional language à la XML [20] with multi-methods, show that the resulting system has decidable minimal types, and discuss subject reduction. Finally, we propose a new object-oriented programming language based on the $\mathrm{ML}_{\leq}$ type system.

1 Introduction

Designing object-oriented extensions of functional languages is a challenging problem which has received much attention lately. Apart from special object-oriented calculi [2], which adopt the view that objects are more primitive than functions, two major approaches have been studied. In the first approach, objects are extensible records with single-dispatch methods attached to them. The major advantage of this approach is that data encapsulation and inheritance are modeled very naturally. However, type systems for extensible records often rely on intricate higher-order formalisms and/or recursive types [11, 33].

The second approach, which has received much less attention both from language designers and type theorists, is to put the emphasis on methods, rather than objects, and resort to module systems to provide scoping and data encapsulation. In this approach, first proposed in CLOS [12], and also used in more recent languages such as Cecil [10], methods are overloaded functions dispatching on the type of all their input arguments simultaneously. Implementing binary operations, such as an addition operator over a hierarchy of numeric classes, is a very natural and easy thing to do in these languages. In contrast, such "multi-methods" are notoriously hard to type and define in the objects-as-records model [7]. Moreover, multi-methods defined by cases look very similar to functions defined by pattern matching, which makes them an obvious candidate for extending functional languages like ML. Despite these advantages, however, no satisfactory static type system has been proposed so far for functional languages with multi-methods.

In this paper, we present what we believe to be the first practical and decidable type system, called $\mathrm{ML}_{\leq}$, suitable for languages that integrate functional programming, parametric polymorphism, class-based object-oriented programming, and higher-order multi-methods. In order to motivate our choices, the examples of this paper are written in an explicitly typed ML-like language, namely, a higher-order functional language with implicit predicative polymorphism, but without type inference. Our key ideas are to introduce subtyping via extensible hierarchies of type constructors, to replace functions defined by pattern matching by methods performing dynamic dispatch on the type of their input arguments, to clearly separate specification and implementation, and to use a module system to provide separate compilation and encapsulation. As a consequence, we do not model implementation inheritance

To appear in the 24th ACM Conf. on Principles of Prog. Languages

Constructor class	Point[ ]	List[$\oplus$]
Type constructors	point blinking cartesian bpol bcart cart	list slist nil cons snil scons
Data types	cart[ ] = $\langle\!\langle\, x\colon$ real ; $y\colon$ real $\rangle\!\rangle$ bpol[ ] = $\langle\!\langle\, r\colon$ real ; $a\colon$ real ; $f\colon$ int $\rangle\!\rangle$ bcart[ ] = $\langle\!\langle\, x\colon$ real ; $y\colon$ real ; $f\colon$ int $\rangle\!\rangle$	nil[α], snil[α] = $\langle\!\langle\ \rangle\!\rangle$ cons[α] = $\langle\!\langle\, h\colon \alpha$; $t\colon$ list[α] $\rangle\!\rangle$ scons[α] = $\langle\!\langle\, h\colon \alpha$; $t\colon$ slist[α] ; $s\colon$ int $\rangle\!\rangle$

Figure 1: The Point and List constructor classes

in the type system itself, but provide this important feature using syntactic sugar.

This extended abstract is organized as follows. In section 2, we give an intuitive introduction to the system using simple examples, motivating our choices as we go along. Section 3 gives a more formal introduction to the system as well as major results (completeness, decidability, and minimal typing). For the sake of simplicity, we restrict ourselves to single-module languages. In section 4, we propose a new modular object-oriented programming language with multi-methods based on the type system of section 3. We conclude in section 5 with a discussion of related work. A complete exposition of the system, including all proofs which had to be omitted due to space restrictions, can be found in [4].

2 Overview

2.1 Constructor classes

Our guiding principle in the design of ML$_\leq$ has been to favor the generalization of well-understood concepts, rather than introducing new and ad-hoc ones. To start with, instead of defining objects as extensible records, we take the view that objects are fixed tagged records, exactly as in ML. For instance, the bottom part of fig. 1 defines three records representing points in the plane with Cartesian coordinates, and blinking points in the plane with polar and Cartesian coordinates. These records are tagged with cart, bpol, and bcart respectively. The same figure also defines records tagged with nil and cons, representing empty and non-empty lists, as well as records tagged with snil and scons, representing empty and non-empty sized lists with constant-time access to their size.

We call cart, bpol, bcart, snil, nil, cons, and scons *data type constructors*. The only run-time entities in ML$_\leq$ are functions, methods, and records tagged with data type constructors. Note that the x fields of cart[ ] and bcart[ ], for instance, are totally unrelated. In other words, we do not model implementation inheritance.

One way to write multi-methods is to define them for every possible combination of tags. However, it is often desirable to define uniform behaviors over a group of tags by a single definition. That is why we introduce the notion of type constructor as a means of naming groups of tags and allowing for the definition of methods uniformly over these groups. Technically, we introduce subtyping through user-defined extensible hierarchies of type constructors like blinking, cartesian, point, slist, and list. As opposed to ML, every data type constructor is a valid type constructor and denotes the group that consists of only that tag.

In order to prevent arbitrary overloading, we group semantically related type constructors into extensible and partially ordered *constructor classes* like Point and List. Monotypes are built from type constructors in the usual way. We identify zero-ary type constructors like point with the monotype point[ ]. The intuition is that a type like blinking denotes the set of all blinking points, irrespective of their representation as data types. The ordering between constructors reflects the inclusion of the sets they denote. Similarly, the monotype cons[int] denotes the type of non-empty lists of integers, whereas $\forall\alpha.$ nil[α] is the type of the empty list. Consequently, extensible classes like List generalize closed algebraic ML datatypes like

$$\textsf{datatype list}[\alpha] \;=\; \textsf{nil} \mid \textsf{cons of } (\alpha \star \textsf{list}[\alpha]);$$

Monotypes are partially ordered by a *structural* subtyping relation based on the *variance* of each class. The variance of a class is a tuple of elements of $\{\oplus, \ominus, \otimes\}$ that specifies the arity of the type constructors of the class as well as the variance of each type parameter. For instance, class List is unary and covariant, since we intend $\mathsf{cons}[\mathsf{int}]$ to be a subtype of $\mathsf{list}[\mathsf{real}]$ (assuming int is a subtype of real). However, the binary class Arrow, which contains the arrow type constructor, is contravariant in its first argument, and covariant in its second argument, so that $\mathsf{real} \to \mathsf{int}$ is a subtype of $\mathsf{int} \to \mathsf{real}$. All data types of a given class must conform to its variance. For instance, the record implementation of $\mathsf{cons}[\alpha]$ is valid because α being covariant both in the h field and in the t field, it is covariant in the record. However, this record could not have an extra field with type $\alpha \to \alpha$, in which α is non-variant. This variance specification allows us to reason about the type constructors of a given class irrespective of the actual contents of the class, which is clearly a prerequisite for extensibility in the context of object-orientation.

The partial ordering of type constructors is arbitrary and allows multiple inheritance of specifications (but not of implementations). The only restriction is that data type constructors be minimal, which, as we shall see in section 2.5, is essential for typing polymorphic methods. The declaration of a datatype like scons implicitly declares a constructor function *scons* with type

$$\forall\alpha.\ \alpha \to \mathsf{slist}[\alpha] \to \mathsf{int} \to \mathsf{scons}[\alpha]$$

that builds a sized cons from an α, a sized list of α's, and the size of the list, as well as three selector functions

$$\begin{aligned} \mathit{scons.h}: &\ \ \forall\alpha.\ \mathsf{scons}[\alpha] \to \alpha \\ \mathit{scons.t}: &\ \ \forall\alpha.\ \mathsf{scons}[\alpha] \to \mathsf{slist}[\alpha] \\ \mathit{scons.s}: &\ \ \forall\alpha.\ \mathsf{scons}[\alpha] \to \mathsf{int} \end{aligned}$$

which, in contrast with ML, are total over their domain.

2.2 Methods

Instead of referring to the notion of a *self* object, as in the objects-as-records paradigm, we propose to define methods as overloaded functions dispatching on the tags of all their arguments simultaneously, in pretty much the same way that ML functions perform pattern matching on the tags of their arguments.

In the simple system presented in this paper, the only patterns allowed are "_" (any), or a type constructor like slist, meaning that for the associated branch to be selected, the tag of the actual argument must be a subconstructor of slist. Strictly speaking, we do not propose a generalization of pattern matching, since ML patterns can be more complex than ours, but our system could be enhanced to allow all ML patterns. In section 4, we show how both dynamic dispatch and pattern matching could be integrated in a real programming language.

For instance, method *head* of fig. 2 raises an exception by default, and returns the h field of conses and

$\mathit{vcons}: \forall\alpha.\ (\alpha, \mathsf{list}[\alpha]) \to \mathsf{list}[\alpha];$
$\mathit{vcons}\ (h\!: \_,\ t\!: \_) = \mathit{cons}\ h\ t;$
$\mathit{vcons}\ (h\!: \_,\ t\!: \mathsf{snil}) = \mathit{scons}\ h\ t\ \mathit{1};$
$\mathit{vcons}\ (h\!: \_,\ t\!: \mathsf{scons}) = \mathit{scons}\ h\ t\ (\mathit{1} + (\mathit{scons.s}\ t));$

$\mathit{head}: \forall\alpha.\ \mathsf{list}[\alpha] \to \alpha;$
$\mathit{head}\ (x\!: \_) = \mathsf{raise}\ \mathit{Empty};$
$\mathit{head}\ (x\!: \mathsf{cons}) = \mathit{cons.h}\ x;$
$\mathit{head}\ (x\!: \mathsf{scons}) = \mathit{scons.h}\ x;$

$\mathit{map}: \forall\alpha_{\mathsf{List}}, \beta, \gamma.\ (\alpha_{\mathsf{List}}[\beta], \beta \to \gamma) \to \alpha_{\mathsf{List}}[\gamma];$
$\mathit{map}\ (l\!: \mathsf{nil},\ f\!: \_) = \mathit{nil};$
$\mathit{map}\ (l\!: \mathsf{snil},\ f\!: \_) = \mathit{snil};$
$\mathit{map}\ (l\!: \mathsf{cons},\ f\!: \_) =$
$\quad \mathit{cons}\ (f\ (\mathit{cons.h}\ l))\ (\mathit{map}\ (\mathit{cons.t}\ l, f));$
$\mathit{map}\ (l\!: \mathsf{scons},\ f\!: \_) =$
$\quad \mathit{scons}\ (f\ (\mathit{scons.h}\ l))\ (\mathit{map}\ (\mathit{scons.t}\ l, f))\ (\mathit{scons.s}\ l);$

Figure 2: Operations on lists

sized conses. As opposed to ML, the order in which the alternatives of a method are defined is irrelevant, since dynamic dispatch is based on a "best match" approach. This choice is of course essential to ensure that alternatives of a given method (assumed to be specified by its type in some interface) can be implemented in several modules. We show in section 3.3 that by imposing that the set of patterns of a method be a *partition* of the domain of the method, it is possible to guarantee the absence of "message not understood" or "match failure" run-time errors (exhaustivity), and to ensure the existence of a best match (non-ambiguity).

Our methods are thus always total over their domain. For instance, method *freq* of fig. 3 is total over its domain $\mathsf{blinking}$, and, as opposed to *head*, this method does not have a "catch-all" alternative to ensure exhaustivity. Such a method could not have been written as it is in ML, where intermediate type constructors like $\mathsf{blinking}$ cannot be defined.

More interesting, method *vcons* of fig. 2 is a "virtual constructor" dispatching on the type of its second argument. This method by default builds a regular cons, except when its second argument is a sized list, in which case it builds a sized cons. Using this virtual constructor, a list built from a sized nil will only consist in sized conses. For instance, the following expression

$$\begin{aligned} E_1 = \mathit{vcons}\,(&\mathit{bpol}\ \mathit{1.0}\ \ \mathit{0.0}\ \ \mathit{1},\\ &\mathit{vcons}\,(\mathit{bcart}\ \mathit{1.0}\ \ \mathit{2.0}\ \ \mathit{3}, \mathit{snil})) \end{aligned}$$

builds a non-empty heterogeneous sized list of polar and Cartesian blinking points. Note that it is not possible to build arbitrary heterogeneous lists in $\mathrm{ML}_{\leq}$, as opposed to type systems based on dynamics [1, 26]. Also, note that our methods allow a form of overloading that is not possible in ML. For instance, it is possible to define a fully polymorphic function like

$$\begin{aligned}&\mathit{shift}\colon \forall\alpha.\ \alpha \to \alpha;\\&\mathit{shift}\ (p\colon \_) = p;\\&\mathit{shift}\ (p\colon \mathsf{cart}) = \mathit{cart}\ ((\mathit{cart.x}\ \ p) + \mathit{1.0})\ (\mathit{cart.y}\ \ p);\end{aligned}$$

which is essentially the identity, except for non-blinking Cartesian points. This function is well typed, because cart being a minimal data type constructor in any extension of class Point, the run-time tag cart of

$$\mathit{cart}\ ((\mathit{cart.x}\ \ p) + \mathit{1.0})\ (\mathit{cart.y}\ \ p)$$

is identical to the tag of p for any non-blinking Cartesian point p. Methods like *shift* can thus be used to perform a weak form of `typecase` statement [1], and can be used in particular to perform narrowing type casts. The scheme we propose to define methods is thus a mixture of what is traditionally called "parametric polymorphism" in the functional programming community[1], and "polymorphism", or "dynamic dispatch", in the object-oriented community.

2.3 Constrained types

The system we have described so far, in which functions and methods are explicitly typed, can be made to work only if every expression has a minimal type. To understand the problem, assume given a function *twice* with type

$$\forall\alpha.\ (\alpha \to \alpha) \to (\alpha \to \alpha)$$

and a function *trunc* with type $\mathsf{real} \to \mathsf{int}$. The ML scheme for typing the application of a polymorphic function like *twice* to an argument like *trunc* consists in applying the most specific monomorphic instance of *twice* to the type of the argument. However, two instances of the type of *twice* can be applied here, namely $(\mathsf{int} \to \mathsf{int}) \to (\mathsf{int} \to \mathsf{int})$ and $(\mathsf{real} \to \mathsf{real}) \to (\mathsf{real} \to \mathsf{real})$, none of which is more specific than the other. We thus propose to type the expression (*twice trunc*) as

$$\forall\alpha\colon (\mathsf{int} \le \alpha \wedge \alpha \le \mathsf{real}).\ \alpha \to \alpha$$

The intuitive denotation of this polymorphic constrained type is, as in ML, a type which is below all its ground instances, that is, all the ground substitutions of $\alpha \to \alpha$ such that α satisfies the constraint $\mathsf{int} \le \alpha \wedge \alpha \le \mathsf{real}$. In other words, an expression with this type has both type $\mathsf{int} \to \mathsf{int}$ and type $\mathsf{real} \to \mathsf{real}$, and can be used in any context where one of these monomorphic types is acceptable.

[1] There are generally very few functions with type $\forall\alpha.\ \alpha \to \alpha$ in traditional models of parametric polymorphism, where methods like *shift* cannot be defined.

$$\begin{aligned}&\mathit{freq}\colon \mathsf{blinking} \to \mathsf{int};\\&\mathit{freq}\ (p\colon \mathsf{bpol}) = \mathit{bpol.f}\ \ p;\\&\mathit{freq}\ (p\colon \mathsf{bcart}) = \mathit{bcart.f}\ \ p;\end{aligned}$$

$$\begin{aligned}&\mathit{move}\colon \forall\alpha\colon \alpha \le \mathsf{point}.\ \alpha \to \alpha;\\&\mathit{move}\ (p\colon \mathsf{cart}) = \mathit{cart}\ (-1.2 \times (\mathit{cart.x}\ p))\\&\qquad\qquad\qquad\qquad (-1.2 \times (\mathit{cart.y}\ p));\\&\mathit{move}\ (p\colon \mathsf{bcart}) = \mathit{bcart}\ (-1.2 \times (\mathit{cart.x}\ p))\\&\qquad\qquad\qquad\qquad (-1.2 \times (\mathit{cart.y}\ p))\ (\mathit{freq}\ p);\\&\mathit{move}\ (p\colon \mathsf{bpol}) = \mathit{bpol}\ (1.2 \times (\mathit{bpol.r}\ p))\\&\qquad\qquad\qquad\qquad (\pi + (\mathit{bpol.a}\ p))\ (\mathit{freq}\ p);\end{aligned}$$

Figure 3: Operations on points

This example shows that polymorphic constrained types (types for short) allow the minimal typing of term application in the context of primitive subtyping. However, many syntactically different types can have the same meaning. For instance, we certainly intend the types $(\forall\,\emptyset.\ \mathsf{int})$ and $\forall\alpha\colon (\mathsf{int} \le \alpha \wedge \alpha \le \mathsf{int}).\ \alpha$ to have the same meaning, since they have the same unique ground instance int. However, not only are we interested in semantic equivalence between types, but we find it useful to define a partial ordering between types, for instance to check that the type of a function in a module conforms to its specification in a recursive let or in some interface. This idea of subtyping polytypes is reminiscent of the subtyping rules of $\mathrm{F}_{\le}$, and departs from the tradition in predicative type systems to rely on a non-deterministic instantiation rule for typing term application.

In order to ensure that the ordering between types is compatible with the ordering between monotypes, we thus say that a polytype τ_2 is a subtype of another polytype τ_1 if every ground instance of τ_1 is above some ground instance of τ_2 w.r.t. the ordering on monotypes. As usual, we say that two types are equivalent if they are subtypes of one another. For instance, $\forall\alpha\colon \mathsf{int} \le \alpha.\ \alpha$ is a subtype of $\forall\,\emptyset.\ \mathsf{real}$, because there exists an α satisfying $\mathsf{int} \le \alpha$ which is above real, namely $\alpha = \mathsf{real}$. As a matter of fact, the former type is equivalent to int, and the above definition of subtyping is compatible with the interpretation of the universal quantifier of a type as a greatest lower bound operator.

2.4 Type application

The general typing rule for term application is the following. Suppose e_1 has type $\tau_1 = \forall\vartheta_1\colon \kappa_1.\ \theta_1 \to \theta_1'$, where ϑ_1 is a list of variables, κ_1 is a constraint, and θ_1 and θ_1' are monotypes with free variables in ϑ_1, and that e_2 has type $\tau_2 = \forall\vartheta_2\colon \kappa_2.\ \theta_2$, with ϑ_1 and ϑ_2 disjoint. Then $(e_1\ e_2)$ has type

$$\mathit{app}(\tau_1, \tau_2) \ = \ \forall\vartheta_1, \vartheta_2\colon \kappa_1 \wedge \kappa_2 \wedge \theta_2 \le \theta_1.\ \theta_1'$$

provided the constraint $\kappa_1 \wedge \kappa_2 \wedge \theta_2 \leq \theta_1$ is satisfiable, as defined in section 3.1. The interpretation of this typing rule is that $(e_1\ e_2)$ has any ground monotype θ_1' such that $\theta_1 \to \theta_1'$ is a ground instance of the type of e_1 and θ_1 is above some ground instance θ_2 of the type of e_2, which is what is intuitively required to apply the function. For instance, the type of $(id\ 1.0)$, where id is the identity with type $\forall\alpha.\ \alpha \to \alpha$, has type $\forall\alpha\colon \mathsf{float} \leq \alpha.\ \alpha$, which is equivalent to $\forall\emptyset.\ \mathsf{float}$. Similarly, the (static) type of expression E_1 of section 2.2 is

$$\begin{aligned}\tau_1 \ = \ \forall\alpha,\beta,\gamma\colon (&\mathsf{bpol} \leq \alpha \wedge \mathsf{list}[\beta] \leq \mathsf{list}[\alpha] \wedge \\ &\mathsf{bcart} \leq \beta \wedge \mathsf{snil}[\gamma] \leq \mathsf{list}[\beta]).\ \mathsf{list}[\alpha]\end{aligned}$$

which, thanks to the variance of List, is in fact formally equivalent to

$$\forall\alpha\colon (\mathsf{bpol} \leq \alpha \wedge \mathsf{bcart} \leq \alpha).\ \mathsf{list}[\alpha]$$

which, in turn, is a subtype of $\tau_2 = \forall\emptyset.\ \mathsf{list}[\mathsf{blinking}]$. An interesting question is whether or not τ_2 is a subtype of τ_1. In a closed world, the only solution of the constraint of τ_1 is $\alpha = \mathsf{blinking}$, which is below $\mathsf{blinking}$, so we could be tempted to consider the two types equivalent. However, in an open world, it may be the case that some module of the program extends class Point and defines a strict subconstructor $\mathsf{sblinking}$ of $\mathsf{blinking}$ above both bpol and bcart. Such an extension, called an admissible extension, is allowed provided it does not modify the ordering between existing type constructors. In the context of the extended class, $\alpha = \mathsf{sblinking}$ is thus a ground instance of τ_1 but is not above the only ground instance $\mathsf{blinking}$ of τ_2. In section 3, we shall define a complete and decidable axiomatization of subtyping based on a notion of constraint implication that is invariant w.r.t. admissible extensions of classes.

In conclusion, the type of E_1 is a strict subtype of $\mathsf{list}[\mathsf{blinking}]$, and can be read as the type of all lists containing blinking points with polar and Cartesian coordinates, but nothing else, and in particular, no other kind of blinking points. In other words, a type like $\forall\alpha\colon (\mathsf{bpol} \leq \alpha \wedge \mathsf{bcart} \leq \alpha).\ \alpha$ can be read as "the smallest α above bpol and bcart" and can be understood as the set union of bpol and bcart.

2.5 Polymorphic multi-methods

In addition to ensuring minimal types, constrained polymorphic types also allow for a very precise typing of methods. For instance, method *move* of fig. 3 has type

$$\forall\alpha\colon \alpha \leq \mathsf{point}.\ \alpha \to \alpha$$

meaning that *move* returns an object with the same tag as its argument of type point. The implementation of

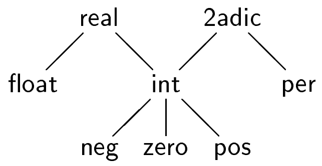

Figure 4: The numeric class Num

move conforms to this specification since bpol, bcart, and cart are minimal data type constructors. Moreover, we show in section 3 that the type of *move* is a strict subtype of $\forall\emptyset.\ \mathsf{point} \to \mathsf{point}$, which shows that *move* can be used anywhere a function with the latter type is expected. More interesting, as opposed to single-dispatch languages, method *move* is a first-class function which can be passed as a parameter to other functions, such as the higher-order polymorphic multimethod *map* of fig. 2. The type of *map* states that given a list with type $\alpha_{\mathsf{List}}[\beta]$, where α_{List} is some type constructor in List, and a function with type $\beta \to \gamma$, *map* returns a list with type $\alpha_{\mathsf{List}}[\gamma]$, which shows in particular that *map* applied to an empty list returns an empty list. Note that this precise type ensures that the recursive call of *map* in the fourth alternative of its definition is a sized list and can thus be used to build a sized cons. Similarly, it can be shown that the expression $E_2 = map\ (E_1,\ move)$ has type

$$\forall\alpha\colon (\mathsf{bpol} \leq \alpha \wedge \mathsf{bcart} \leq \alpha \wedge \alpha \leq \mathsf{point}).\ \mathsf{list}[\alpha]$$

which shows that $freq\ (head\ E_2)$ is well-typed and has type int. Methods like *move* can be specified in some object-oriented languages with a specification like

```
abstract class point is
    virtual method move(): like self
end
```

However, the advantage of our approach is that it generalizes to multi-methods, and is also much more expressive. For instance, the subtraction method *sub* of fig. 5 has type $\forall\alpha\colon \mathsf{int} \leq \alpha.\ (\alpha,\alpha) \to \alpha$. The hierarchy used in this example, shown in fig. 4, has positive, negative, and zero integers, as well as floating point numbers and periodic 2-adic numbers[2]. The type of *sub* ensures that per's and float's cannot be subtracted from one another (since the constraint $\mathsf{int} \leq \alpha \wedge \mathsf{per} \leq \alpha \wedge \mathsf{float} \leq \alpha$ has no solution over the class shown in fig. 4) and also ensures that the static type of a subtraction is always a supertype of int. For instance, using the typing rule for application, we can show that the type of $sub\ (+1, -2)$ is $\forall\alpha\colon (\mathsf{int} \leq \alpha \wedge \mathsf{pos} \leq \alpha \wedge \mathsf{neg} \leq \alpha).\ \alpha$, which is equivalent to $\forall\emptyset.\ \mathsf{int}$. In other words, the type of *sub* is polymorphic above int, and constant below int.

[2] This numeric hierarchy is used in the hardware description language $2z$, which motivated this work [40].

It is interesting to compare our approach to similar systems in the literature. The language presented by Reppy and Riecke [35, 36] allows the definition of methods returning objects with the type `selfty` of the receiver, so that methods like *move* can be written. However, the restriction that `selfty` cannot occur in negative positions in the type of methods prevents methods like *sub* to be given the precise type of our example. The language of Bruce et al. [6] uses a similar notion of *MyType* without the restriction on negative positions. However, the interpretation of *MyType* in a method type like *MyType* → *MyType* and the interpretation of the type variable α in the type $\forall\alpha\colon \mathsf{int} \leq \alpha.\ (\alpha, \alpha) \to \alpha$ of the *sub* function are quite different: *MyType* refers to the dynamic type of the receiver (i.e., the first argument of the multi-method), whereas α is the "minimum" type which is above *both* arguments of the method. This interpretation of *MyType* thus prevents reals to be subtracted from integers. One possible remedy, for single-inheritance languages, could be to replace the notion of `selfty` or *MyType* by the notion of `alike`, denoting the smallest supertype of all arguments with type `alike`, including the receiver.

Type classes [19, 31, 41] and constructor classes [23] have also been advocated as a means to provide some of the functionality of methods in ML-like languages. In essence, these systems allow the instantiation of *overloaded specifications* which consist in type templates with a single type variable, such as the instantiation of the template $(\alpha, \alpha) \to \alpha$ for $\alpha = \mathsf{int}$ and $\alpha = \mathsf{real}$. In the absence of any subtyping relation between int and real, such simple specifications cannot express complex types such as the type of the *sub* method and, in particular, do not allow the typing of mixed operations like $sub(1.2, 3)$. Multi-parameter type classes, i.e., type templates with more than one variable, have thus been proposed to lift this restriction. The idea is to use a template like $(\alpha, \beta) \to \gamma$ and instantiate it with all possible interesting combinations of α, β, and γ, for example $(\mathsf{int}, \mathsf{int}, \mathsf{int}), (\mathsf{int}, \mathsf{real}, \mathsf{real})$. However, multi-parameter type classes are not without problems: type-checking is undecidable in general [15], it is possible to overload functions with structurally different signatures, and the overloading resolution algorithm can be quite tricky and unintuitive, a mixture which has already proven very dangerous in C++. This is why we believe that the dispatching mechanism based on structural subtyping that is used in $\mathrm{ML}_{\leq}$ is closer in spirit to that of classical object-oriented languages than the non-structural overloading scheme used in languages like Haskell and Gofer. Nonetheless, it may be interesting to add type classes to $\mathrm{ML}_{\leq}$ to allow for overloaded functions like *print* with type $\forall\alpha\colon Print(\alpha).\ \alpha \to \mathsf{unit}$ where *Print* is an inductively defined predicate on types.

$$
\begin{aligned}
&toFloat\colon \mathsf{real} \to \mathsf{float};\\
&toPer\colon \mathsf{2adic} \to \mathsf{per};\\
&subInt\colon (\mathsf{int}, \mathsf{int}) \to \mathsf{int};\\
&subFloat\colon (\mathsf{float}, \mathsf{float}) \to \mathsf{float};\\
&subPer\colon (\mathsf{per}, \mathsf{per}) \to \mathsf{per};\\[1ex]
&sub\colon \forall\alpha\colon \mathsf{int} \leq \alpha.\ (\alpha, \alpha) \to \alpha;\\
&sub\ (x_1\colon \mathsf{float},\ x_2\colon \mathsf{real}) = subFloat\ (x_1,\ toFloat\ x_2);\\
&sub\ (x_1\colon \mathsf{per},\ x_2\colon \mathsf{2adic}) = subPer\ (x_1,\ toPer\ x_2);\\
&sub\ (x_1\colon \mathsf{int},\ x_2\colon \mathsf{int}) = subInt\ (x_1,\ x_2);\\
&sub\ (x_1\colon \mathsf{int},\ x_2\colon \mathsf{float}) = subFloat\ (toFloat\ x_1,\ x_2);\\
&sub\ (x_1\colon \mathsf{int},\ x_2\colon \mathsf{per}) = subPer\ (toPer\ x_1,\ x_2);
\end{aligned}
$$

Figure 5: Subtraction

3 Type system

We now formally define our type system. A *constructor class* C is given by a name, a finite and non-empty set $\mathcal{T}_C$ of *type constructors* t_C, a subset $\mathcal{D}_C \subseteq \mathcal{T}_C$ of *data type constructors* d_C, a partial order $\sqsubseteq_C$ on $\mathcal{T}_C$ such that data type constructors are minimal with respect to $\sqsubseteq_C$, and a tuple of elements of the set $\{\oplus, \ominus, \otimes\}$ called the *variance* of the class. The arity of a class is the length of its variance.

A *type structure* is a finite set $\mathcal{T}$ of constructor classes with distinct names and pairwise disjoint sets of type constructors. We assume that every type structure contains a $(\ominus, \oplus)$-variant class Arrow with at least one data type constructor $\to$. In order to model object-orientation, we must provide for the extension of type structures. Adding new classes is never a problem (assuming that there are no name clashes), because type constructors of different classes are completely unrelated to each other. When new type constructors are added to existing classes, one has to preserve the ordering on existing type constructors as well as the minimality of data type constructors. We thus formally define $\mathcal{T}^*$ to be an *admissible extension* of $\mathcal{T}$ if for every class C in $\mathcal{T}$, there exists a class C^* in $\mathcal{T}^*$ with the same name and variance such that $\mathcal{T}_{C^*}$ is a superset of $\mathcal{T}_C$, the intersection of $\mathcal{T}_C$ and $\mathcal{D}_{C^*}$ is equal to $\mathcal{D}_C$, and for all type constructors $t_1, t_2 \in \mathcal{T}_C$, we have $t_1 \sqsubseteq_{C^*} t_2$ if and only if $t_1 \sqsubseteq_C t_2$. The overall requirement that data type constructors are minimal in any class implies that $\mathcal{T}^*$ cannot define a subconstructor of any data type constructor defined in $\mathcal{T}$.

We assume given countable and pairwise disjoint sets of *type variables* v, v', etc., and, for each class C, *C-constructor variables* v_C, v'_C, etc. The set of monotypes θ over $\mathcal{T}$ is the least set containing type variables such that when Θ_C is a list of monotypes whose length agrees with the arity of C, and ϕ_C is a *C-constructor*, i.e., a C-type constructor t_C in $\mathcal{T}_C$ or a C-constructor variable

v_C, then $\phi_C[\Theta_C]$ is a monotype. A variable-free monotype is said to be $\mathcal{T}$-ground (or just ground, for short). As usual, we write $\theta_1 \to \theta_2$ instead of $\to[\theta_1, \theta_2]$.

The ordering $\leq$ on ground monotypes is the least relation such that $t_C \sqsubseteq_C t'_C$ and $\Theta_C \leq_C \Theta'_C$ imply $t_C[\Theta_C] \leq t'_C[\Theta'_C]$, where the relation $\leq_C$ on lists of ground monotypes is defined as the componentwise ordering induced by the variance of C. For instance, $\theta_1, \theta_2 \leq_{\mathsf{Arrow}} \theta'_1, \theta'_2$ if and only if $\theta'_1 \leq \theta_1$ and $\theta_2 \leq \theta'_2$. We use ϑ to denote a list of type or constructor variables, and ϑ_C to denote a list of distinct type variables whose length agrees with the arity of C.

3.1 Constraints

A constraint κ is a conjunction of inequalities $\phi_C \sqsubseteq \phi'_C$ between C-constructors and inequalities $\theta \leq \theta'$ between monotypes. A variable-free constraint is said to be ground. We treat constraints as sets of conjuncts, and we write $\kappa\{\kappa'\}$ to denote that κ' is a subset of κ. We denote by $\theta = \theta'$ the constraint $\theta \leq \theta' \wedge \theta' \leq \theta$, and by *true* the empty constraint.

Intuitively, we intend a constraint to be satisfiable if it has a solution on ground monotypes. However, for the subtyping relation between polytypes discussed in section 2.3, we also need a notion of satisfiability of a constraint w.r.t. some other. To this end, we define the implication of constraint κ_2 by constraint κ_1 for all ϑ as the judgment $\forall\vartheta.\ \kappa_1 \models \kappa_2$ axiomatized by the rules of fig. 6 together with the transitivity rule

$$\frac{\forall\vartheta.\ \kappa_1 \models \kappa_2 \qquad \forall\vartheta.\ \kappa_2 \models \kappa_3}{\forall\vartheta.\ \kappa_1 \models \kappa_3}\ [\mathit{Trans}]$$

In rule *VElim*, we write $\theta \simeq \theta'$ to denote either $\theta \leq \theta'$ or $\theta' \leq \theta$. A ϑ-substitution σ maps type variables to monotypes and C-constructor variables to C-constructors, and is the identity over variables in ϑ. We denote by $\kappa[\sigma]$ the application of σ to κ.

Rule *VIntro* introduces new variables on the right-hand side by abstracting away certain subterms of the left-hand side. For example, an instance of this rule is

$$\begin{array}{ll} \forall\emptyset. & \mathsf{bpol} \leq \mathsf{blinking} \wedge \mathsf{bcart} \leq \mathsf{blinking} \\ \models & \mathsf{bpol} \leq \alpha \wedge \mathsf{bcart} \leq \alpha \end{array}$$

which reads "$\mathsf{bpol} \leq \mathsf{blinking}$ and $\mathsf{bcart} \leq \mathsf{blinking}$ implies the existence of some α such that $\mathsf{bpol} \leq \alpha$ and $\mathsf{bcart} \leq \alpha$." In other words, the free variables of κ_1 and κ_2 that are not in ϑ are existentially quantified.

Rules *MIntro*, *MElim*, and *VElim* reflect the fact that the ordering on monotypes is purely structural, that is, comparable types must have the same "shape". Therefore, every solution for a variable v in constraint $v \simeq \phi_C[\Theta_C]$ is of the form $\phi'_C[\Theta'_C]$.

$$\begin{array}{lll}
[\mathit{Approx}] & \forall\vartheta.\ \kappa\{\kappa'\} \models \kappa' & \\
[\mathit{CRef}] & \forall\vartheta.\ \kappa \models \kappa \wedge \phi_C \sqsubseteq \phi_C & \\
[\mathit{CTrans}] & \forall\vartheta.\ \kappa\{\phi_C \sqsubseteq \phi'_C \sqsubseteq \phi''_C\} \models \kappa \wedge \phi_C \sqsubseteq \phi''_C & \\
[\mathit{CGnd}] & \forall\vartheta.\ \kappa \models \kappa \wedge t_C \sqsubseteq t'_C \quad (\mathit{if}\ t_C \sqsubseteq_C t'_C) & \\
[\mathit{CMin}] & \forall\vartheta.\ \kappa\{\phi_C \sqsubseteq d_C\} \models \kappa \wedge d_C \sqsubseteq \phi_C & \\
[\mathit{MRef}] & \forall\vartheta.\ \kappa \models \kappa \wedge \theta \leq \theta & \\
[\mathit{MTrans}] & \forall\vartheta.\ \kappa\{\theta \leq \theta' \leq \theta''\} \models \kappa \wedge \theta \leq \theta'' & \\
[\mathit{MIntro}] & \forall\vartheta.\ \kappa\{\Theta_C \leq_C \Theta'_C \wedge \phi_C \sqsubseteq \phi'_C\} & \\
 & \models\ \ \kappa \wedge \phi_C[\Theta_C] \leq \phi'_C[\Theta'_C] & \\
[\mathit{MElim}] & \forall\vartheta.\ \kappa\{\phi_C[\Theta_C] \leq \phi'_C[\Theta'_C]\} & \\
 & \models\ \ \kappa \wedge \phi_C \sqsubseteq \phi'_C \wedge \Theta_C \leq_C \Theta'_C & \\
[\mathit{VIntro}] & \forall\vartheta.\ \kappa[\sigma] \models \kappa \quad (\mathit{if}\ \sigma\ \mathit{is\ a}\ \vartheta\text{-}\mathit{substitution}) & \\
[\mathit{VElim}] & \forall\vartheta.\ \kappa\{v \simeq \phi_C[\Theta_C]\} & \\
 & \models\ \ \kappa \wedge v = v_C[\vartheta_C] \quad (v_C, \vartheta_C\ \mathit{fresh}) &
\end{array}$$

Figure 6: Constraint implication

We say that a constraint κ is *well-formed* if and only if the judgment $\forall\emptyset.\ \mathit{true} \models \kappa$ is derivable. The following theorems show that implication is decidable, and that the axiomatization of fig. 6 is both sound and complete w.r.t. the extensibility of constructor classes. In particular, it follows that well-formedness and satisfiability coincide.

Theorem 1 *If κ_1 is a well-formed constraint, and κ_2 is an arbitrary constraint, then it is decidable whether $\forall\vartheta.\ \kappa_1 \models \kappa_2$ holds. In particular, well-formedness of constraints is decidable.*

Theorem 2 *Let ϑ be a list of variables, and κ_1 and κ_2 be two well-formed constraints. Then $\forall\vartheta.\ \kappa_1 \models \kappa_2$ is derivable if and only if for every admissible extension $\mathcal{T}^*$ of $\mathcal{T}$ and every $\mathcal{T}^*$-ground substitution σ_1 such that $\kappa_1[\sigma_1]$ is satisfied in $\mathcal{T}^*$, there exists a $\mathcal{T}^*$-ground substitution σ_2 that agrees with σ_1 on the variables of ϑ such that $\kappa_2[\sigma_2]$ is a ground constraint satisfied in $\mathcal{T}^*$.*

The algorithm to determine the well-formedness of a constraint κ described in [4] first checks that κ is well-kinded and then determines a representation of its solutions in the form of a most general substitution constrained by a set of independent base constraints on type variables and C-constructors for each class C.

A constraint κ is well-kinded if the set of equations built from κ by replacing C-constructors by the uninterpreted function symbol C and inequality symbols by equalities is unifiable. Similar notions have been introduced in the literature [17, 29, 30].

Given a well-kinded constraint κ, the first step of the decision procedure for well-formedness consists in

$$
\begin{array}{ll}
(N_1) & \kappa,\ (\kappa' \wedge \phi_C[\Theta_C] \leq \phi'_C[\Theta'_C]) \ \longrightarrow \\
& \qquad \kappa,\ (\kappa' \wedge \phi_C \sqsubseteq \phi'_C \wedge \Theta_C \leq_C \Theta'_C) \\
(N_2) & \kappa,\ \kappa' \{v \simeq \phi_C[\Theta_C]\} \ \longrightarrow \\
& \qquad (\kappa \wedge v = v_C[\vartheta_C]),\ \kappa'[v_C[\vartheta_C]/v]
\end{array}
$$

Figure 7: Normalization of well-kinded constraints

rewriting (true, κ) by the two rules of fig. 7, where v_C and ϑ_C are assumed to be fresh. The well-kindedness of κ ensures that this process eventually terminates, and that the result $(\kappa^{=}, \kappa^{\leq})$ is such that $\kappa^{=}$ represents a "most general substitution", whereas $\kappa^{\leq}$ is a conjunction of a constraint on type variables and constraints κ_C between C-constructors for each class C. The last step for deciding well-formedness consists in checking that each constraint κ_C is satisfiable over the current type structure, which is trivially decidable by finite enumeration. For instance, the solutions of the constraint

$$(\alpha \leq \beta \rightarrow \gamma) \ \wedge \ (\gamma \leq \mathsf{real})$$

are of the form $\alpha = \alpha_1 \rightarrow \alpha_2[\,]$ and $\gamma = \gamma_1[\,]$ where $\beta \leq \alpha_1$ and $\alpha_2 \sqsubseteq \gamma_1 \wedge \gamma_1 \sqsubseteq \mathsf{real}$, which is satisfiable, for instance, taking $\alpha_2 = \mathsf{int}$ and $\gamma_1 = \mathsf{real}$.

The procedure for deciding $\forall\vartheta.\ \kappa_1 \models \kappa_2$ consists in applying the decision procedure for well-formedness to compute the most general substitution $\kappa_1^{=}$ of κ_1 and the base constraints $\kappa_1^{\leq}$ as above. For the sake of simplicity, we assume that κ_1 and κ_2 do not share variables except for the ones in ϑ. It can be shown that the free variables of $\kappa_1^{\leq}$, together with the partial ordering induced by $\kappa_1^{\leq}$, define the "most general" admissible extension of the current type structure satisfying κ_1. Deciding the implication then amounts to deciding the well-formedness of κ_2 w.r.t. this extended type structure, thus considering the variables of κ_1 as constants.

It is easy to see that rewriting may cause an exponential increase in the size of the constraints. Since satisfiability of a set of base constraints is NP-complete [27, 34], our decision procedure for well-formedness is thus at worst doubly exponential, but [38] shows that the problem is in fact in DEXPTIME. Finally, it follows from [37] that deciding well-formedness is PSPACE-hard. A fortiori, deciding implication is PSPACE-hard.

However, when type-checking real-life programs, we believe that the rewriting step will not cause a blowup in the number of variables, and the only costly part of the decision procedure will be to test the satisfiability of base constraints, which is NP-complete. Figure 8 gives a fixpoint-based algorithm C-SAT, inspired by an incomplete algorithm by Fuh and Mishra [17], to decide the satisfiability of a base constraint κ_C. The initial call $C\text{-SAT}(\kappa_C, \sigma_C)$ must be performed with a valuation function σ_C mapping every C-constructor ϕ_C to the set

```
procedure C-SAT(κ_C, σ_C) is
  let σ'_C = ⋂_{i≥0} Φ^i_C(κ_C, σ_C) in
    if ∃ v_C. |σ'_C(v_C)| = 0 then fail;
    if ∀ v_C. |σ'_C(v_C)| = 1 then succeed;
    for each v_C such that |σ'_C(v_C)| > 1 do
      for each t_C in σ'_C(v_C) do
        C-SAT(κ_C, σ'_C[v_C ↦ {t_C}])
end C-SAT
```

Figure 8: Satisfiability of base constraints

$\mathcal{T}_C$. The functional Φ_C is defined as follows

$$
\begin{array}{rcl}
\Phi_C(\kappa_C, \sigma_C)(t_C) & = & \{t_C\} \\
\Phi_C(\kappa_C, \sigma_C)(v_C) & = & \bigcap_{\kappa_C\{\phi_C \sqsubseteq v_C\}} \uparrow_C \sigma_C(\phi_C) \\
& \cap & \bigcap_{\kappa_C\{v_C \sqsubseteq \phi_C\}} \downarrow_C \sigma_C(\phi_C)
\end{array}
$$

where $\uparrow_C S$ (resp. $\downarrow_C S$) denotes the upper (resp. lower) ideal generated by the subset S of $\mathcal{T}_C$ w.r.t. the partial order $(\mathcal{T}_C, \sqsubseteq_C)$. Our experience with a first implementation of this algorithm has been very encouraging.

3.2 Types and domains

Typing judgments in our system are always expressed w.r.t. a *constraint context* $\Delta = (\vartheta\colon \kappa)$, which is used to store type information for symbols defined in the context of the current declaration, and intuitively asserts the existence of variables ϑ satisfying κ. We say that a type $\tau_1 = \forall\vartheta_1\colon \kappa_1.\ \theta_1$ is well-formed w.r.t. Δ if ϑ and ϑ_1 are disjoint and the free variables of κ_1 and θ_1 are either in ϑ or in ϑ_1, and if κ implies κ_1 for all ϑ. The latter condition ensures that κ_1 is well-formed for every solution of the "global" constraint κ. As a concrete example, consider the function

$$\mathsf{fun}\ \{\alpha \mid \mathit{true}\}\ (x\colon \alpha) \Rightarrow \mathsf{let}\ y = \mathit{sub}(x, 1)\ \mathsf{in}\ (\mathit{not}\ x)$$

expressed in the language defined in section 3.3. If, in the definition of well-formed types, we simply required that $\kappa \wedge \kappa_1$ be well-formed, instead of requiring that κ imply κ_1 for all ϑ, then y would have type

$$\forall\beta\colon (\mathsf{int} \leq \beta \wedge \alpha \leq \beta \wedge \mathsf{pos} \leq \beta).\ \beta$$

w.r.t. constraint context $(\alpha\colon \mathit{true})$, and $(\mathit{not}\ x)$ would have type $\forall\emptyset\colon \alpha \leq \mathsf{bool}.\ \mathsf{bool}$, so that the function would appear to be well-typed, with type $\forall\alpha\colon \alpha \leq \mathsf{bool}.\ \alpha \rightarrow \mathsf{bool}$, but every application of that function would fail at run-time.

For the sake of simplicity, we assume that the bound variables of types can be freely α-converted to names that do not occur in the constraint context. In the light of theorem 2 and of the discussion of section 2.3, we say that $\tau_2 = \forall\vartheta_2 \colon \kappa_2.\ \theta_2$ is a subtype of τ_1 w.r.t. Δ (assuming ϑ_1 and ϑ_2 disjoint), written $\Delta \vdash \tau_2 \leq \tau_1$, if and only if

$$\forall\vartheta, \vartheta_1.\ \kappa \wedge \kappa_1 \models \kappa_2 \wedge \theta_2 \leq \theta_1$$

This definition of subtyping can be seen as a generalization of the instance relation between well-typings of Mitchell [29, 30]. We show in [4] that this rule is also sound w.r.t. the three variants $\forall$-orig, $\forall$-top, and $\forall$-Fun of the subtyping rule of $F_{\leq}$ considered in [9].

It follows immediately from theorem 1 that subtyping is decidable. As an example, let us prove that the type

$$\forall\alpha \colon \alpha \leq \mathsf{point}.\ \alpha \rightarrow \alpha$$

of method *move* is a strict subtype of $\forall\,\emptyset.\ \mathsf{point} \rightarrow \mathsf{point}$ w.r.t. the empty context. We have to prove

$$\forall\emptyset.\ \mathit{true} \models \alpha \leq \mathsf{point} \wedge \alpha \rightarrow \alpha \leq \mathsf{point} \rightarrow \mathsf{point}$$

which follows by rule *VIntro* from

$$\forall\emptyset.\ \mathit{true} \models \mathsf{point} \leq \mathsf{point}\ \wedge \\ \mathsf{point} \rightarrow \mathsf{point} \leq \mathsf{point} \rightarrow \mathsf{point}$$

On the other hand, $\forall\,\emptyset.\ \mathsf{point} \rightarrow \mathsf{point}$ is not a subtype of $\forall\alpha \colon \alpha \leq \mathsf{point}.\ \alpha \rightarrow \alpha$, so the latter type is a strict subtype of the former one. For otherwise, we would have to show

$$\forall\alpha.\ \alpha \leq \mathsf{point} \models \mathsf{point} \rightarrow \mathsf{point} \leq \alpha \rightarrow \alpha$$

which, by rule *MElim*, amounts to proving

$$\forall\alpha.\ \alpha \leq \mathsf{point} \models \alpha = \mathsf{point}$$

which is obviously not derivable.

One distinguishing feature of our system is that the type application operator *app* is monotonic in both arguments w.r.t. the subtyping relation, which shows that every functional type $\tau_1 = \forall\vartheta_1 \colon \kappa_1.\ \theta_1 \rightarrow \theta_1'$ can be identified with a monotonic type transformer, as opposed to $F_{\leq}$ where type application is defined in terms of syntactic substitution. Note that the downside of this property is that type application in $\mathrm{ML}_{\leq}$ is approximated [4].

We formally define the domain $\mathit{dom}(\tau_1)$ of the functional type τ_1 as $\delta_1 = \exists\vartheta_1 \colon \kappa_1.\ \theta_1$. Intuitively, domains denote downward closed sets of types. It can be shown that *dom* is contravariant w.r.t. the subtyping relation. We say that a domain $\delta_2 = \exists\vartheta_2 \colon \kappa_2.\ \theta_2$ is a subdomain of δ_1 w.r.t. Δ (assuming ϑ_1 and ϑ_2 disjoint), written $\Delta \vdash \delta_2 \leq \delta_1$, if and only if

$$\forall\vartheta, \vartheta_2.\ \kappa \wedge \kappa_2 \models \kappa_1 \wedge \theta_2 \leq \theta_1$$

and we say that a type $\tau_2 = \forall\vartheta_2 \colon \kappa_2.\ \theta_2$ belongs to δ_1 w.r.t. Δ, written $\Delta \vdash \tau_2 \in \delta_1$, if

$$\forall\vartheta.\ \kappa \models \kappa_1 \wedge \kappa_2 \wedge \theta_2 \leq \theta_1$$

It can be shown that $\mathit{app}(\tau_1, \tau_2)$ is well-formed w.r.t. Δ if and only if τ_2 belongs to $\mathit{dom}(\tau_1)$ w.r.t. Δ. Moreover, the subtyping and membership relations are transitive in the following sense

$$\frac{\Delta \vdash \tau_1 \leq \tau_2 \quad \Delta \vdash \tau_2 \in \delta_3}{\Delta \vdash \tau_1 \in \delta_3} \qquad \frac{\Delta \vdash \tau_1 \in \delta_2 \quad \Delta \vdash \delta_2 \leq \delta_3}{\Delta \vdash \tau_1 \in \delta_3}$$

so that every subtype of a type in $\mathit{dom}(\tau_1)$ also belongs to $\mathit{dom}(\tau_1)$, which agrees with the substitutivity principle of object-oriented languages and justifies the view of domains as downward closed sets of types.

Finally, we show in [4] that types form a preorder, and that two compatible types (i.e., types with a common upper bound) $\forall\vartheta_1 \colon \kappa_1.\ \theta_1$ and $\forall\vartheta_2 \colon \kappa_2.\ \theta_2$ have the following least upper bound

$$\forall v, \vartheta_1, \vartheta_2 \colon (\kappa_1 \wedge \kappa_2 \wedge \theta_1 \leq v \wedge \theta_2 \leq v).\ v$$

where v is a fresh type variable, and ϑ_1 and ϑ_2 are assumed to be disjoint. Dually, two compatibles domains δ_1 and δ_2 always have a greatest lower bound $\delta_1 \wedge \delta_2$.

3.3 Type checking

Fig. 9 gives type checking rules for an explicitly typed functional language à la XML [20] with higher-order multi-methods. Programs consist in a single expression type checked w.r.t. a fixed type structure $\mathcal{T}$ that we assume to be defined with some concrete syntax. A well-formed typing context is a pair $(\Delta; \Gamma)$ where Δ is a well-formed constraint context, and Γ is a list of bindings of the form $x \colon \tau$ for expression variables, where each τ is well-formed w.r.t. Δ. We assume that Γ binds constructor and selector functions to their type, as defined at the end of section 2.1. The domain $\delta = \exists\vartheta \colon \kappa.\ \theta$ of a function

$$\mathsf{fun}\ \{\vartheta \mid \kappa\}\ (x \colon \theta) \Rightarrow e$$

is given explicitly but, as opposed to methods, the return type of functions is inferred from their bodies by rule *Fun*. A method m is an expression of the form[3]

$$\mathsf{meth}\ \{\vartheta \mid \kappa\}\ (x \colon \theta) \colon \theta' \Rightarrow [\pi_1 \Rightarrow e_1; \ldots; \pi_n \Rightarrow e_n]$$

where each *pattern* π_i is a special kind of domain of the form[4] $\exists\vartheta.\ (\theta_1, \ldots, \theta_n)$, where ϑ is a list of type variables

[3] We used some syntactic sugar, explained in section 4, for the examples of section 2.

[4] We assume that tuples are implicitly defined data types, with appropriate constructor and selector functions.

$$\Delta; \Gamma\{x\colon \tau\} \vdash x\colon \tau \quad [Var]$$

$$\frac{\Delta; \Gamma \vdash e\colon \tau \qquad \Delta \vdash \tau \leq \tau'}{\Delta; \Gamma \vdash e\colon \tau'}\ [Sub] \qquad \frac{\Delta; \Gamma \vdash e_1\colon \tau_1 \qquad \Delta; \Gamma[x_1\colon \tau_1] \vdash e_0\colon \tau_0}{\Delta; \Gamma \vdash (\mathsf{let}\ x_1 = e_1\ \mathsf{in}\ e_0)\colon \tau_0}\ [Let]$$

$$\frac{\Delta; \Gamma[x_1\colon \tau_1, \ldots, x_n\colon \tau_n] \vdash e_i\colon \tau_i \qquad (0 \leq i \leq n)}{\Delta; \Gamma \vdash (\mathsf{letrec}\ x_1\colon \tau_1 = e_1; \ldots; x_n\colon \tau_n = e_n\ \mathsf{in}\ e_0)\colon \tau_0}\ [Letrec]$$

$$\frac{\Delta; \Gamma \vdash e\colon \tau \qquad \Delta; \Gamma \vdash e'\colon \tau' \qquad \Delta \vdash \tau' \in \mathit{dom}(\mathit{fun}(\tau))}{\Delta; \Gamma \vdash (e\ e')\colon \mathit{app}(\mathit{fun}(\tau), \tau')}\ [App]$$

$$\frac{\Delta[v, \vartheta\colon \kappa \wedge v \leq \theta]; \Gamma[x\colon \forall\emptyset.\ v] \vdash e\colon (\forall\vartheta'\colon \kappa'.\ \theta') \quad (v\ \mathit{fresh})}{\Delta; \Gamma \vdash (\mathsf{fun}\ \{\vartheta \mid \kappa\}\ (x\colon \theta) \Rightarrow e)\colon (\forall v, \vartheta, \vartheta'\colon \kappa \wedge \kappa' \wedge v \leq \theta.\ v \to \theta')}\ [Fun]$$

$$\frac{\begin{array}{c}\delta = \exists\vartheta\colon \kappa.\ \theta \qquad \pi_i = \exists\vartheta_i.\ \theta_i \qquad \pi_1, \ldots, \pi_n \text{ is a partition of } \delta \text{ w.r.t. } \Delta \\ \Delta[v, \vartheta, \vartheta_i\colon \kappa \wedge v \leq \theta \wedge v \leq \theta_i]; \Gamma[x\colon \forall\emptyset.\ v] \vdash e_i\colon (\forall\emptyset.\ \theta') \quad (1 \leq i \leq n,\ v\ \mathit{fresh})\end{array}}{\Delta; \Gamma \vdash (\mathsf{meth}\ \{\vartheta \mid \kappa\}\ (x\colon \theta)\colon \theta' \Rightarrow [\pi_1 \Rightarrow e_1; \ldots; \pi_n \Rightarrow e_n])\colon (\forall\vartheta\colon \kappa.\ \theta \to \theta')}\ [Meth]$$

Figure 9: Typing rules

with at most one occurrence in $(\theta_1, \ldots, \theta_n)$, and each θ_i is either a single variable or $t_C[\vartheta_C]$ for some type constructor t_C and variables ϑ_C.

We think of a method as a set of functions, one for each pattern π_i, whose type is a subtype of the method type, restricted to the domain π_i. In contrast to ML patterns, which may be complex, the present definition of ML$_{\leq}$ patterns allows for dynamic dispatch according to the outermost type constructor only. Rule *Meth* defines the type of method m as $\forall\vartheta\colon \kappa.\ \theta \to \theta'$, provided that two conditions are met.

First, the set of patterns must be a *partition* of the method's domain $\exists\vartheta\colon \kappa.\ \theta$, to ensure the absence of "method not understood" errors at run-time, as well as the existence of a most specific pattern for every type in the domain of the method. Technically, we say that a set of patterns $\pi_1, \ldots, \pi_n$ is a partition of $\delta = \exists\vartheta\colon \kappa.\ (\theta_1, \ldots, \theta_k)$ if (1) every pattern is compatible with δ w.r.t. Δ, and (2) for all data type constructors $d_{C_1}, \ldots, d_{C_k}$ in the current type structure such that $\delta[\Delta]$ and

$$\pi \;=\; \exists\vartheta_{C_1}, \ldots, \vartheta_{C_k}.\ (d_{C_1}[\vartheta_{C_1}], \ldots, d_{C_k}[\vartheta_{C_k}])$$

are compatible, the set $\{\pi_i \mid \exists i \in [1,n].\ \pi \leq \pi_i\}$ has a minimum element. In the presence of a module system, this decidable condition must be checked at link time, when all the data type constructors, and therefore, all the entities that can possibly exist at run-time, are known. The closure $\delta[\Delta]$ of δ w.r.t. $\Delta = (\vartheta'\colon \kappa')$ is defined by $\exists\,\vartheta', \vartheta\colon \kappa' \wedge \kappa.\ (\theta_1, \ldots, \theta_k)$

Second, the body e_i of each alternative must have type $\forall\emptyset.\ \theta'$ in the context where x is assumed to have both monotype θ enforced by the domain of m and monotype θ_i enforced by pattern π_i.

The remaining rules are straightforward. Rule *Sub* is a subsumption rule reminiscent of F$_{\leq}$. Rule *App* uses the *app* and *dom* operators defined earlier, as well as the upper-closure operator *fun* defined by

$$\mathit{fun}(\forall\vartheta\colon \kappa.\ \theta) \;=\; \forall v, v', \vartheta\colon (\kappa \wedge \theta \leq v \to v').\ v \to v'$$

Assuming that e and e' have minimal types, the covariance of *fun* and *app*, the contravariance of *dom*, and the transitivity of the subtyping and the membership relations ensure that $(e\ e')$ has a minimal type. This is the main argument in the proof of the following theorem, which expresses that ML$_{\leq}$ has minimal types (but not minimal typings, since we are not concerned about type inference in this paper).

Theorem 3 *Let $(\Delta; \Gamma)$ be a well-formed typing context. It is decidable whether an expression e is well-typed in the context $(\Delta; \Gamma)$. If e is well-typed, it has a minimal type and this minimal type can be effectively determined.*

Having minimal types is an important property of type systems. However, soundness is even more important. For lack of space, we omit the definition of an operational semantics for ML$_{\leq}$, and the proof of soundness. A strict operational semantics, as well as a subject-reduction theorem, can be found in the technical report [4]. This operational semantics tags every run-time object with its minimal, closed type τ, and dynamic dispatch is performed by selecting the smallest pattern π such that τ belongs to $\delta[\Delta] \wedge \pi$.

```
interface List is

  // Covariant class of all list constructors
  class List<covariant T>;

  // Lists (declares list)
  abstract list in List<T> is
    // No field
  with
    // Constructors
    vcons(h: T): cons<T>;
    cons(h: T): #cons<T>;
    scons(h: T): #scons<T>;

    // Methods
    head(): T;
    tail(): list<T>;
    size(): int;
    reverse(): alike;
    concat(l: alike): alike;
    map(f: T -> U): L<U>
       where alike = L<T> end
  end;

  // Empty list (declares nil and #nil)
  concrete nil < list in List<T> is
    // No new field or method
  end;

  // Cons lists (declares cons and #cons)
  concrete cons < list in List<T> is
    // Head and tail fields
    h: T;
    t: list<T>
  end;

  // Sized lists (declares slist)
  abstract slist < list in List<T> is
    // No new field or method
  end;

  // Empty sized lists (declares snil and #snil)
  concrete snil < nil, slist in List<T> is
    // No new field or method
  end;

  // Sized cons lists (declares scons and #scons)
  concrete scons < cons, slist in List<T> is
    // Size field
    s: int
  end

end List;
```

```
module List is

  open List;

  // Constructors
  list::vcons(h: _) = self.cons(h);
  slist::vcons(h: _) = self.scons(h);
  list::cons(h: _) =
     #cons {h=h, t=self};
  list::scons(h: _) =
     #scons {h=h, t=self, s=1+self.size()};

  // Head of a list
  nil::head() = raise Empty;
  cons::head() = self.h;

  // Tail of a list
  nil::tail() = raise Empty;
  cons::tail() = self.t;

  // Size of a list
  nil::size() = 0;
  cons::size() = 1+self.t.size();
  scons::size() = self.s;

  // Reverse method and local auxiliary method rev
  nil::reverse() = self;
  #cons::reverse() =
     rev(self.t, #nil.cons(self.h)});
  #scons::reverse() =
     rev(self.t, #snil.scons(self.h)});

  rev(l: list<T>, r: L<T>): L<T>
     where L <: cons end;
  rev(l: nil, r: _) = r;
  rev(l: cons {h, t}, r: #cons) =
     rev(t, r.cons(h));
  rev(l: cons {h, t}, r: #scons) =
     rev(t, r.scons(h));

  // Concatenation
  nil::concat(l: _) = l;
  #cons::concat(l: _) =
     self.t.concat(l).cons(self.h);
  #scons::concat(l: _) =
     self.t.concat(l).scons(self.h);

  // Map method
  #nil::map(f: _) = #nil;
  #snil::map(f: _) = #snil;
  #cons::map(f: _) =
     self.t.map(f).cons(f(self.h));
  #scons::map(f: _) =
     self.t.map(f).scons(f(self.h))

end List;
```

Figure 10: The List package

4 Towards a real programming language

We now sketch a new class-based object-oriented language with multi-methods, and show how its syntax can be desugared into $\mathrm{ML}_{\leq}$. Instead of formally defining the language, we exemplify its constructs at the hand of the `List` package of fig. 10. Note that the type hierarchy of this package, shown in fig. 11, is more refined than the one of fig. 1.

First of all, we show how to add implementation inheritance. For the sake of simplicity, as for most classical object-oriented languages, we do not separate the implementation inheritance hierarchy from the subtyping hierarchy, but doing so would be easy. A declaration like `abstract list` declares a type constructor `list` in constructor class `List`. This type constructor corresponds to an abstract parameterized class in OO parlance. A declaration like `concrete cons` declares both a type constructor `cons` below `list` in class `List` and a data type constructor `#cons` below `cons` in class `List`. As always, `cons` can have subconstructors, but `#cons` is minimal, and has its own constructor and selector functions. The declaration of `cons` imposes that every data type constructor below `cons`, including `#cons`, has two fields `h` of type `T` and `t` of type `list<T>`. The syntax `#cons {h=1, t=#nil}` can be used to build a `#cons` data object, and access, e.g., to the `h` field of a cons object is performed via a method named `List::h` implicitly defined as follows

```
List::h(self: cons<T>): T;
List::h(self: #cons) = #cons.h(self);
List::h(self: #scons) = #scons.h(self);
```

using the selector functions of each data type below `cons`. Note the use of constructor classes to provide a scope for field and method names. Also, note that the above notation for defining methods by cases, which allows a method to be implemented in different modules, is syntactic sugar for the following $\mathrm{ML}_{\leq}$ method

$$\begin{array}{l}\mathsf{meth}\ \{\alpha \mid \mathit{true}\}\ (\mathit{self}\colon \mathsf{cons}[\alpha])\colon \alpha \Rightarrow \\ [\quad \exists\beta.\ \#\mathsf{cons}[\beta] \Rightarrow \#\mathsf{cons}.h\ \ \mathit{self}; \\ \quad\ \exists\gamma.\ \#\mathsf{scons}[\gamma] \Rightarrow \#\mathsf{scons}.h\ \ \mathit{self} \quad]\end{array}$$

If `l` is a list with a type in the domain of method `List::h`, the dot notation `l.h` translates into the function call `List::h(l)`. Such a disambiguation, based on the class of the first argument, is always possible provided that `l` does not have the empty type $\forall\alpha.\ \alpha$.

Class methods like `map` specify a regular method `List::map` with two parameters: an implicit `self` parameter with type `alike`, and a parameter `f` with type `T -> U`. The fact that `map` is defined in the scope of the definition of `list` automatically enforces the constraint `alike <: list<T>`. The type of `map` is thus

$$\forall\, T, U, L_{\mathsf{List}}, \mathit{alike}\colon \mathit{alike} \leq \mathsf{list}[T] \wedge \mathit{alike} = L_{\mathsf{List}}[T].\ (\mathit{alike}, T \to U) \to L_{\mathsf{List}}[U]$$

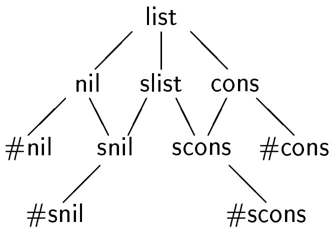

Figure 11: Type hierarchy of the `List` package

As explained in section 2.5, `alike` does not necessarily denote the minimum type of the `self` parameter, in contrast to what `selfty` does in Object ML [35, 36]. For example, method `concat` has type

$$\forall\, T, \mathit{alike}\colon \mathit{alike} \leq \mathsf{list}[T].\ (\mathit{alike}, \mathit{alike}) \to \mathit{alike}$$

which shows, in particular, that the concatenation of two empty lists is an empty list, a property not expressible in any language we are aware of. The dot notation `l.map(f)` can be used as for fields to perform the function call `List::map(l, f)`.

A class method like `head` is implemented by defining method `List::head` by cases, and the syntax

```
cons::head() = self.h
```

in the definition of `head` is syntactic sugar for

```
List::head(self: cons) = self.h
```

that makes the `self` parameter explicit. The latter style of definition offers the possibility to perform pattern matching on the `self` parameter as in

```
List::head(self: cons {h}) = h
```

where `cons {h}` is syntactic sugar for `cons {h=h}`.

Note the use of the local method `rev` to implement `reverse`. This method accepts any `list<T>` as first argument, any `L<T>`, where `L` is a type constructor below `cons`, as second argument, and returns a `L<T>`. Defining methods independently from the type hierarchy allows this kind of methods to be defined as required without the need to introduce a new subtype to hold the method. Also, note how the fairly imprecise type of the virtual constructor `vcons` prevents its use in methods like `map` with very precise types. Finally, remark that an interface defines a name space for the entities that it declares. For instance, the qualified name of class `List` is `List.List`, and the fully qualified name of method `map` is `List.List::map`. The meaning of a program with several modules and interfaces consists in a global letrec containing all the declarations and implementations contained in these modules and interfaces.

5 Related work and conclusion

Our interest in this paper has been to enhance the standard Hindley-Milner type system [21, 28] for an explicitly typed version of ML so that it can be used for higher-order object-oriented languages with polymorphic multi-methods. We believe that $ML_{\leq}$ is a practical and natural extension of the Hindley-Milner type system, and that constraint implication is a unifying concept for such extensions. In particular, it should not be too difficult to add type classes [31, 41] to our framework by refining the notion of constraint implication. We conjecture that type inference for $ML_{\leq}$ programs without methods could be easily adapted from techniques developed in the literature [3, 16, 17, 18, 22, 30, 32]. We believe that inferring the type of methods will be much more challenging.

The model developed in this paper is very similar to that of the programming language Cecil [10], in particular by its distinction between concrete and abstract classes, the distinction between subtyping and implementation inheritance, the use of method specifications, and the use of modules to provide encapsulation. However, the type system proposed by Chambers and Leavens is only first-order and monomorphic, and the specification of methods by means of sets of monomorphic signatures is less expressive and more ad-hoc than ours. Nonetheless, many of the techniques developed in [10] could be adapted to our system, in particular techniques for true separate compilation of multi-methods and compilation of dynamic dispatch.

Castagna et al. [8] have defined an extension of $F_{\leq}$ that allows function overloading in a higher-order setting with explicit polymorphism and primitive subtyping. Their model is quite powerful but technically rather tricky, as all impredicative models. Moreover, methods lack specifications, which can be a problem for modularity and scalability.

$ML_{\leq}$ also has strong links with all systems derived from the Hindley-Milner type system, in particular, systems of overloaded functions built around the notions of type and constructor classes [23, 24, 41]. These systems are incomparable to $ML_{\leq}$ in terms of expressive power (non-structural overloading vs. true methods) but we find $ML_{\leq}$ much closer in spirit to class-based object-oriented languages, and also easier to extend.

Duggan [13, 14], and then Odersky, Wadler, and Wehr [31] have proposed the use of kinded types, which are polymorphic constrained types with constraints on available instances of the operations used by function bodies. This approach can be made to work under the "open world" assumption [13, 31], but types are rather hard to read, since they mention program functions, and lead to method specifications which are dependent on the program's text, which may be a problem for modularity and scalability. On the other hand, type inference is made easier by such an approach.

Kaes [25] has tackled the decidability of type inference in the context of overloading, subtyping, and recursive types, using polymorphic constrained types which are more expressive than ours, and with a precise typing of arithmetic operators. Moreover, his notion of "structural similarity" is fairly close to our notion of constructor class. However, his paper does not address the problem of defining methods and performing dynamic dispatch.

Mitchell [29, 30], Fuh and Mishra [17, 18], Aiken and Wimmers [3], and Eifrig, Smith, and Trifonov [16], have also addressed the problem of type inference in the presence of primitive subtyping. Our notion of constraint implication can be seen as a generalization of the instance notion for well-typings. Smith et al. [16] propose a record-based object-oriented language with polymorphic methods which are less expressive than ours, and the use of recursive types leads to a complex and potentially undecidable subtyping relation with an incomplete, but decidable, axiomatization [39].

The model proposed by Reppy and Riecke [35, 36] is record-based and single dispatch, and is powerful enough to type a method like *move*, but not to type a multi-method like *sub*. Moreover, their model lacks parameterized classes. In contrast with our hypothesis that data type constructors be minimal, the technique used by Reppy and Riecke to implement methods like *move* that return a new object with exactly the type `selfty` of the receiver is to pass the constructor `new` of the receiver as an argument.

Mitchell and Jategaonkar [22] propose to extend ML pattern matching with flexible records and primitive subtyping, in order to allow some form of object-oriented programming. Their system only has built-in operations with constrained types like that of method *sub*. However, they informally show how to accommodate a user-defined class hierarchy of points with methods like $move: \forall\, t \subseteq \mathsf{point}.\ t \rightarrow t$ with a unique implementation defined in the root class.

Finally, we want to mention that a type checker for $ML_{\leq}$ has been implemented in Objective Caml. This prototype type checks approximately 400 lines per second on a low-cost workstation. A system derived from $ML_{\leq}$ will be used in a forthcoming version of the hardware description language $2z$ developed in collaboration with Gérard Berry and Jean Vuillemin [40].

References

[1] M. Abadi, L. Cardelli, B. Pierce, G. Plotkin. Dynamic Typing in a Statically-Typed Language. ACM Transactions on

Programming Languages and Systems, 13(2) (1991) 237–268

[2] M. Abadi, L. Cardelli. A Theory of Primitive Objects: Second-Order Systems. Proc. of the European Symposium on Programming, Springer-Verlag (1994) 1–25

[3] A. Aiken, E. Wimmers. Type Inclusion Constraints and Type Inference. Proceedings FPCA'93 (1993) 31–41

[4] F. Bourdoncle, S. Merz. On the integration of functional programming, class-based object-oriented programming, and multi-methods. Technical Report 26, Centre des Mathématiques Appliquées, École des Mines de Paris (1996) `http://www.ensmp.fr/~bourdonc/`

[5] F. Bourdoncle, S. Merz. Primitive subtyping $\wedge$ implicit-polymorphism $\models$ object-orientation. Third International Workshop on Foundations of Object-Oriented Languages (1996) `http://www4.informatik.tu-muenchen.de/~merz/`

[6] K. B. Bruce, A. Schuett, R. van Gent. PolyTOIL: a type-safe polymorphic object-oriented language (extended abstract). Proc. of ECOOP'95, LNCS 952 (1995) 27–51

[7] K. B. Bruce, L. Cardelli, G. Castagna, The Hopkins Object Group, G. T. Leavens, B. C. Pierce. On binary methods. Technical report LIENS-95-14 (1995)

[8] G. Castagna, G. Ghelli, G. Longo. A calculus for overloaded functions with subtyping. Information and Computation, 117(2) (1995) 115–135

[9] G. Castagna, B. C. Pierce. Corrigendum: Decidable Bounded Quantification. Proc. of the 22nd Symp. on Principles of Programming Languages (1995) 408–408

[10] C. Chambers, G. Leavens. Typechecking and Modules for Multi-Methods. Technical Report UW-CS TR 95-08-05, University of Washington (1995)

[11] P. L. Curien, G. Ghelli. Coherence of subsumption, minimum typing and the type checking of $F_{\leq}$. Mathematical Structures in Computer Science 2(1) (1992)

[12] L. G. DeMichiel, R. P. Gabriel. Common lisp object system overview. ECOOP'87, LNCS 276 (1987) 151–170

[13] D. Duggan, J. Ophel. Kinded Parametric Overloading. Technical Report CS-94-35, University of Waterloo (1994)

[14] D. Duggan. Polymorphic Methods With Self Types for ML-like Languages. Technical Report CS-95-03, University of Waterloo (1995)

[15] D. Duggan, J. Ophel. Multi-Parameter Parametric Overloading. Technical report, University of Waterloo (1995) (submitted to publication)

[16] J. Eifrig, S. Smith, V. Trifonov. Sound Polymorphic Type Inference for Objects. Proc. of OOPSLA'95 (1995) 169–184

[17] Y.-C. Fuh, P. Mishra. Type inference with Subtypes. 2nd European Symp. on Programming, LNCS 300 (1988) 94–114

[18] Y.-C. Fuh, P. Mishra. Polymorphic Subtype Inference: Closing the Theory-Practice Gap. TAPSOFT'89, LNCS 352 (1988) 167–183

[19] K. Hammond, editor. Report on the Programming Language Haskell, version 1.3 (1995)

[20] R. Harper, J. Mitchell. On the Type Structure of Standard ML. TOPLAS 15(2) (1993) 211–252

[21] R. Hindley. The principal type-scheme of an object in combinatory logic. Trans. Amer. Math. Soc., 146 (1969) 29–60

[22] L. Jategaonkar, J. C. Mitchell. Type Inference with extended pattern matching and subtypes. Fundamenta Informaticae. 19 (1, 2) (1993) 127–166

[23] M. P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. FPCA'93 (1993)

[24] S. Kaes. Parametric Polymorphism. Proc. of 2nd European Symp. on Programming, LNCS 300 (1988)

[25] S. Kaes. Type inference in the presence of overloading, subtyping and recursive types. Proc. of Conf. on Lisp and Functional Programming (1992) 193–204

[26] X. Leroy, M. Mauny. Dynamics in ML. Journal of Functional Programming, 3(4) (1993) 109–122

[27] P. Lincoln, J. C. Mitchell, Algorithmic Aspects of Type Inference with Subtypes. Proc. of the 19th ACM Symp. on Principles of Programming Languages (1991) 293–304.

[28] R. Milner. A theory of type polymorphism in programming. Journal of Computer and System Sciences, vol. 17 (1978) 348–375

[29] J. C. Mitchell. Coercion and Type Inference (Summary). Proc. of the 11th ACM Symp. on Principles of Programming Languages (1984) 175–185

[30] J. C. Mitchell. Type inference with simple subtypes. Journal of Functional Programming, 1(3) (1991) 245–285

[31] M. Odersky, P. Wadler, M. Wehr. A second look at overloading. Proc. of the 7th Conf. on Functional Programming and Computer Architecture (1995) 135–146

[32] J. Palsberg. Efficient inference of object types. Proc. IEEE Symp. Logic in Computer Science (1994) 186–195

[33] B. C. Pierce, D. N. Turner. Simple type-theoretic foundations for object-oriented programming. Journal of Functional Programming 4 (2) (1994) 207–247

[34] V. Pratt, J. Tiuryn. Satisfiability of Inequalities in a Poset. Technical Report 95-15(215), Institute of Informatics, Warsaw University (1995)

[35] J. Reppy, J. Riecke. Simple objects for Standard ML. Proc. of the 1996 SIGPLAN Conference on Programming Languages Design and Implementation (1996) 171–180

[36] J. Reppy, J. Riecke. Classes in Object ML via Modules. Presented at the Third International Workshop on Foundations of Object-Oriented Languages (1996) `http://www.cs.williams.edu/~kim/FOOL/`

[37] J. Tiuryn. Subtype Inequalities. Proceedings of the Seventh Symposium on Logic in Computer Science (1992) 308–315

[38] J. Tiuryn, M. Wand, Type Reconstruction with Recursive Types and Atomic Subtyping. 18th Colloquium on Trees in Algebra and Programming (1993)

[39] V. Trifonov, S. Smith. Subtyping Constrained Types. Third International Static Analysis Symposium. Lecture Notes in Computer Science 1145 (1996) 349–365

[40] J. Vuillemin. On circuits and numbers. IEEE Trans. on Computers, 43:8 (1994) 868–879

[41] P. Wadler, S. Blott. How to make ad-hoc polymorphism less ad-hoc. Proc. of the 16th ACM Symp. on Principles of Programming Languages (1989) 60–76