Refinement Types for TLA$^+$

Stephan Merz[1] and Hernán Vanzetto[1,2]

[1] INRIA, Villers-lès-Nancy, France & LORIA
[2] Microsoft Research-INRIA Joint Centre, Saclay, France

Abstract. TLA$^+$ is a specification language, mainly intended for concurrent and distributed systems. Its non-temporal fragment is based on a variant of (untyped) ZF set theory. Motivated by the integration of the TLA$^+$ Proof System with SMT solvers or similar tools based on multi-sorted first-order logic, we define a type system for TLA$^+$ and we prove its soundness. The system includes refinement types, which fit naturally in set theory. Combined with dependent function types, we obtain type annotations on top of an untyped specification language, getting the best of both the typed and untyped approaches. After implementing the type inference algorithm, we show that the resulting typing discipline improves the verification capabilities of the proof system.

1 Introduction

The specification language TLA$^+$ [11] combines a variant of Zermelo-Fraenkel (ZF) set theory for the description of the data manipulated by algorithms and linear-time temporal logic for the specification of their behavior. The TLA$^+$ Proof System (TLAPS) integrates different backends for automatic proving to provide proof support for TLA$^+$. The work reported here is motivated by the development of an SMT backend through which users of TLAPS interact with standard SMT (satisfiability modulo theories) solvers for non-temporal reasoning in the set theory of TLA$^+$.

In line with the foundations of classical mathematics, TLA$^+$ is an untyped formalism [12]. On the other hand, it is generally accepted that strong type systems such as Martin-Löf type theory or HOL (Church's simple type theory) and its variants help provide semi-automatic proof support for highly expressive modeling languages. Automatic first-order theorem provers, including SMT solvers, are generally based on multi-sorted first-order logic that have interpreted operators over distinguished sorts, such as arithmetic operators over integers. Similarly, specification languages such as Z [19] or B [1] use typed variants of set theory that correspond naturally to multi-sorted first-order logic [5].

A sound way of encoding TLA$^+$ in SMT-LIB [4], the de-facto standard input language for SMT solvers, described in our previous work [14], is to introduce a distinguished sort U corresponding to TLA$^+$ values, with injections from existing sorts, such as $int2u : \mathsf{Int} \to \mathsf{U}$ for integer values. To represent an operator such as addition, we declare a function $plus$ that takes arguments and returns results in U, but we relate it to the built-in addition operator $+$, over the image of $int2u$, by the axiom

$$\forall m, n : \mathsf{Int}.\ plus(int2u(m), int2u(n)) = int2u(m + n).$$

```
1 declare int2u : (Int) U
2 declare plus : (U U) U
3 assert ∀m, n : Int. int2u(m) = int2u(n) ⇒ m = n
4 assert ∀m, n : Int. plus(int2u(m), int2u(n)) = int2u(m + n)
5 assert ¬(∀x : U. (∃n : Int. x = int2u(n)) ⇒ plus(x, int2u(0)) = x)
```

Fig. 1. Encoding of the proof obligation $\forall x.\ x \in Int \Rightarrow x + 0 = x$ in SMT-LIB.

With this representation, the SMT backend will be unable to prove the TLA$^+$ formula $\forall x.\, x + 0 = x$ because the value of the bound variable x is not known to be in the image of $int2u$. Indeed, this formula is not a theorem of TLA$^+$; for example, the expression $\{\} + 0$ is syntactically correct, but its value is unspecified. However, the TLA$^+$ formula $\forall x.\ x \in Int \Rightarrow x + 0 = x$ can be proved, based on the (pretty-printed) SMT-LIB encoding shown in Fig. 1. As can be seen from this example, this style of encoding requires a substantial number of quantified formulas that degrade the performance of SMT solvers. In particular, the hypothesis $x \in Int$ in the TLA$^+$ formula gives rise to the subformula $\exists n : \mathsf{Int}.\ x = int2u(n)$. If we could detect appropriate type information from the original TLA$^+$ formula, we could simply translate it to $\forall x : \mathsf{Int}.\ x + 0 = x$.

The above example motivates the definition of a type system and an associated type inference algorithm for TLA$^+$. Our previous work [14] contained a preliminary proposal in this direction. By necessity, type systems impose restrictions on the admissible formulas, and one can therefore not expect type inference to succeed for all TLA$^+$ proof obligations. If no meaningful types can be inferred, the translation can fall back to the "untyped" encoding described above. The question is then how expressive the type system should be in order to successfully handle a large class of TLA$^+$ formulas. The type system of [14] was fairly restricted and could in certain cases not express adequate type information. In particular, handling function applications in TLA$^+$ often requires precise type information, where it must be proved that the argument is in the domain of the function. For example, consider the TLA$^+$ formula[3]

$$\forall f \in [\{1, 2, 3\} \to Int].\, f[0] < f[0] + 1$$

This formula should not be provable: since 0 is not in the domain of f, we should not infer that $f[0]$ is an integer. In our previous work, we over-approximated the type of f as a function from Int to Int, then generated a side condition that attempted to prove $0 \in \mathsf{dom}\, f$. However, computing the domain of a function is not always as easy as in this example, leading to failed proof attempts. The design of an appropriate type system is further complicated by the fact that some formulas, such as $f[x] \cup \{\} = f[x]$, are actually valid irrespectively of whether $x \in \mathsf{dom}\, f$ holds or not. This observation motivates the use of a more expressive type system. Using refinement types [7, 20], the type of $\mathsf{dom}\, f$ is $\{x : \mathsf{Int} \mid x = 1 \vee x = 2 \vee x = 3\}$. During type inference, the system will try to prove that $x = 0 \Rightarrow x \in \mathsf{dom}\, f$, and this will fail, hence the translation will fall back to the untyped encoding (which will in turn fail to prove the formula, as

[3] In TLA$^+$, $[S \to T]$ denotes the set of functions with domain S and co-domain T, and the application of function f to argument e is written $f[e]$.

it should). In many practical examples, the domain condition can be established during type inference, leading to shorter and simpler SMT proof obligations.

The main contribution of this paper is thus a novel use of refinement types for TLA$^+$ formulas. Since TLA$^+$ is very close to untyped Zermelo-Fraenkel set theory, we believe that our approach is more widely applicable for theorem proving in set-theoretic languages. A type system with refinement types is very expressive and actually quite close to set theory; it gives rise to proof obligations that are undecidable. Specifically, subtyping between two refinement types $\{x : \tau \mid \phi_1\}$ and $\{x : \tau \mid \phi_2\}$ reduces to prove $\phi_1 \Rightarrow \phi_2$. This is comparable to the use of predicate types in the PVS theorem prover [18] where type checking conditions may be generated that have to be discharged interactively. In our case, we divide the problem of type inference into constraint generation and constraint solving. Constraint generation rules are derived directly from type checking rules, and always succeed. For constraint solving, we again use SMT solvers, which may succeed or not. In case constraint solving fails, we fall back to the untyped encoding (restricted to the corresponding part of the proof obligation), which is comparable to dynamic type checking.

Paper outline. In Section 2 we present a formal definition of a fragment of TLA$^+$. Section 3 contains the definition of the type system, including the key concepts of typing hypothesis and safe types, the typing rules and finally the proof of soundness of the system. The typing rules give raise to the inference algorithm in Section 4. Next, we show some experimental results of a prototype implementation of the system in Section 5 and Section 6 concludes.

2 A fragment of TLA$^+$

We now introduce a fragment of TLA$^+$, called $\mathcal{L}$, that represents the essential concepts of TLA$^+$. The main simplifications are: we restrict the discussion to unary operators and do not handle TLA$^+$'s CHOOSE operator, tuples, strings, records, or sequences. In order to adhere to a more standard presentation of ZF set theory, we also assume a distinction between terms (non-Boolean expressions) and formulas, whereas TLA$^+$ does not. However, in the "liberal interpretation" of TLA$^+$ [11] that underlies TLAPS, the results of Boolean connectives are always Boolean. Using a pre-processing step of "Boolification" that replaces all possibly non-Boolean arguments e of Boolean operators by $e = \mathsf{true}$, the distinction between terms and formulas can be recovered.

Syntax. We assume given non-empty, infinite, and disjoint sets $\mathcal{V}$ of variables and $\mathcal{O}$ of (unary) operator symbols, the latter subdivided into Boolean operators w^b and non-Boolean operators w.[4] The set-theoretic kernel of $\mathcal{L}$ is given by the following grammar where for clarity we distinguish between different syntactic categories of expressions.

[4] TLA$^+$ operator symbols correspond to the standard function and predicate symbols of first-order logic but we reserve the term "function" for functional values in TLA$^+$.

Language $\mathcal{L}$ grammar

(terms)	t	$::=$	$v \mid w(e)$
(sets)	s	$::=$	$t \mid \{\} \mid \{e, e\} \mid \mathbb{P}s \mid \cup s \mid \{v \in s : \phi\}$
(expressions)	e	$::=$	s
(formulas)	ϕ	$::=$	$w^b(e) \mid \mathsf{false} \mid \phi \Rightarrow \phi \mid \forall v.\, \phi \mid e = e \mid e \in s$

A term is either a variable symbol v in $\mathcal{V}$ or results from the application of an operator symbol w in $\mathcal{O}$ to an expression. Since TLA$^+$ is a set-theoretic language, every term denotes a set. The language also contains explicit set constructors corresponding to the empty set, pairs, the powerset, the generalized union, and set comprehension. Initially, expressions are just sets. Formulas are built from the application of a Boolean operator symbol w^b to an expression, from false, implication and universal quantification (from which the remaining first-order connectives can be defined), and from the binary operators $=$ and $\in$. This language, plus an object of infinity (the set of integer numbers Int that we will add later), corresponds to MacLane set theory, which is a suitable fragment to formalize large parts of mathematics.

As a first extension of this purely set-theoretic language, we now introduce (total) functions. In standard set theory, functions are defined as binary relations (i.e., sets of pairs) restricted so that each element of the domain is mapped to a unique element in the range of the relation. TLA$^+$ instead introduces functions axiomatically using three primitive constructs. The expression $f[e]$ denotes the result of applying the function f to the expression e, and $\mathsf{dom}\, f$ denotes the domain of f. The expression $\lambda x \in S.\, e$ denotes the function f with domain S such that $f[x] = e$, for any $x \in S$. For $x \notin S$, the value of $f[x]$ is unspecified. The expression $[S \to T]$ denotes the set of functions with domain S and co-domain T. The characteristic predicate for a TLA$^+$ value being a function is defined as $IsAFcn(f) \triangleq f = \lambda x \in \mathsf{dom}\, f.\, f[x]$.

Furthermore, $\mathcal{L}$ also contains arithmetic expressions. Natural numbers are primitive symbols, Int denotes the set of integer numbers, and the operators $+$, $-$, and $<$ denote the usual operations when applied to integers. For further reading, a more detailed presentation of the formal definition of TLA$^+$ appears in [11, Sec. 16].

Extension with functions

(terms)	t	$::=$	$\ldots \mid f[e]$
(sets)	s	$::=$	$\ldots \mid \mathsf{dom}\, f \mid [s \to s]$
(functions)	f	$::=$	$t \mid \lambda v \in s.\, e$
(expressions)	e	$::=$	$\ldots \mid f$

Extension with arithmetic

(sets)	s	$::=$	$\ldots \mid Int$
(numbers)	n	$::=$	$t \mid 0 \mid 1 \mid 2 \mid \cdots \mid n + n \mid n - n$
(expressions)	e	$::=$	$\ldots \mid n$
(formulas)	ϕ	$::=$	$\ldots \mid n < n$

A many-sorted version of $\mathcal{L}$, written $\mathcal{L}^\tau$, is obtained by decorating variables with sorts, and by assigning a type $\langle \tau_1, \tau_2 \rangle$ to every operator where τ_1 and τ_2 denote the type of the argument and of the result. Our type system will be introduced in Section 3. In particular, we will write $\forall v : \tau.\, \phi$ for a quantified formula where the bound variable has sort τ.

The definitions of free variables and substitution are the usual ones for first-order logic over the set of variables $\mathcal{V}$. We write $fv(\phi)$ for the set of free variables of ϕ, and $e[y \leftarrow z]$ for the expression or formula e where all occurrences of the free variable y are substituted by z.

Semantics. A single-sorted *model* $\mathcal{M}$ is composed of a non-empty set $\mathcal{D}$ called the *domain*, a *valuation* function $\varphi : \mathcal{V} \to \mathcal{D}$ that assigns to each variable an element in the domain, and an *interpretation* function $\mathcal{I}$ that, in particular, assigns to each operator symbol w a function $\mathcal{I}(w) : \mathcal{D} \to \mathcal{D}$. The definition of the interpretation continues in the standard way. In particular, models respect the extensionality and foundations axioms of ZF, functions are governed by the axiom

$$\begin{array}{rcl} f = \lambda x \in s.\ e & \Leftrightarrow & \wedge\ \mathit{IsAFcn}(f) \\ & & \wedge\ \mathsf{dom}\ f = s \\ & & \wedge\ \forall y \in s.\ f[y] = e[x \leftarrow y] \end{array}$$

and arithmetic expressions are interpreted in the standard way when arguments are integers. The semantics of the multi-sorted language $\mathcal{L}^\tau$ is analogous with the usual modifications corresponding to the presence of sorts [13].

A formula ϕ is *valid* (noted $\vdash \phi$) iff it holds in every model.

3 A Type System with Refinements

Types are given by the following grammar.

$$\tau ::= \mathsf{t}_1 \mid \mathsf{t}_2 \mid \ \ldots \ \mid \mathsf{Bool} \mid \mathsf{Int} \mid \alpha \mid \mathsf{Set}\,\tau \mid (v : \tau) \to \tau \mid \tau \uplus \tau \mid \{x : \tau \mid \phi\}$$

The basic types consist of a denumerable set of atomic types $\mathsf{t}_1, \mathsf{t}_2, \ldots$, as well as of types Bool for formulas and Int for integers. Further type constructors are directly correlated to set objects. For instance, the Set constructor determines the level of set strata for $\mathbb{P}$ and $\cup$. Type variables α are interpreted over the resulting Herbrand universe of types. A *ground assignment* σ is a total function σ of type variables to atomic types.

A refinement type $\{x : \tau \mid \phi\}$ is intended for representing set comprehension objects. It describes the set of values of type τ that satisfy the refinement predicate ϕ, where x is free in ϕ. Refinement types have the property (3.1) that the refinement of a refinement type is also a refinement type. From this property, we know also that any type τ can be written as the (trivial) refinement type $\{x : \tau \mid \mathsf{true}\}$.

$$\{x : \{y : \tau \mid \phi_1\} \mid \phi_2\} = \{x : \tau \mid \phi_1[y \leftarrow x] \wedge \phi_2\} \tag{3.1}$$

The type of the empty set is defined as the type $\varnothing_\tau \triangleq \mathsf{Set}\,\{x : \tau \mid \mathsf{false}\}$, for any type τ. A pair $\{a, b\}$ has the type $\tau_a \uplus \tau_b$, the logical union of the types of a and b. The union type constructor $\uplus$ is an operation on refinements and sets and it is defined by:

$$\{x : \tau \mid \phi_1\} \uplus \{x : \tau \mid \phi_2\} = \{x : \tau \mid \phi_1 \vee \phi_2\} \tag{3.2}$$

$$(\mathsf{Set}\,\tau_1) \uplus (\mathsf{Set}\,\tau_2) = \mathsf{Set}\,(\tau_1 \uplus \tau_2) \tag{3.3}$$

A function f has the dependent type $(x : \tau_1) \to \tau_2$ [2], where τ_1 represents the domain of f and the term x may occur in the range type τ_2. The variable x of type τ_1 is bound in type τ_2. If x does not occur in τ_2, we can omit it from the syntax to obtain the standard function type $\tau_1 \to \tau_2$.

3.1 Typing Propositions and Typing Hypotheses

When encoding a multi-sorted language into a single-sorted one, the traditional method [6] is straightforward. For every sort τ, it defines a characteristic proposition $\mathcal{P}_\tau$ that represents the set of values having sort τ. For instance, the proposition associated to $\mathsf{Set}\,\tau$ is derived from the axiom of power set. Then, it relativizes the quantifiers, that is, it replaces the sort annotations $x : \tau$ by new hypotheses $\mathcal{P}_\tau(x)$. This method is applied to formulas without type variables, therefore all types should be grounded. For each atomic type $\mathtt{t}_i$, we introduce a new unary predicate symbol t_i and an axiom stating that these predicates partition the universe of ground types in disjoint sets.

Definition 1 (Typing propositions). *Given a type assignment $x : \tau$, an encoding of it can be constructed into the formula $\mathcal{P}_\tau(x)$, defined as follows:*

$$\begin{array}{c}
\mathcal{P}_{\mathtt{t}_i}(x) \triangleq t_i(x) \quad \mathcal{P}_{\mathsf{Bool}}(x) \triangleq x \in \{\mathsf{true}, \mathsf{false}\} \quad \mathcal{P}_{\mathsf{Int}}(x) \triangleq x \in \mathit{Int} \\
\mathcal{P}_{\mathsf{Set}\,\tau}(x) \triangleq \forall z \in x.\, \mathcal{P}_\tau(z) \quad \mathcal{P}_{\tau_1 \uplus \tau_2}(x) \triangleq \mathcal{P}_{\tau_1}(x) \vee \mathcal{P}_{\tau_2}(x) \\
\mathcal{P}_{\{y:\tau \mid \phi\}}(x) \triangleq \mathcal{P}_\tau(x) \wedge \phi[y \leftarrow x] \\
\mathcal{P}_{(x:\tau_1)\to\tau_2}(f) \triangleq \wedge f = \lambda x \in \mathsf{dom}\, f.\, f[x] \\
\qquad\qquad\qquad\qquad \wedge \forall z.\, z \in \mathsf{dom}\, f \Leftrightarrow \mathcal{P}_{\tau_1}(z) \\
\qquad\qquad\qquad\qquad \wedge \forall z.\, \mathcal{P}_{\tau_1}(z) \Rightarrow (\forall x. \mathcal{P}_{\tau_1}(x) \Rightarrow \mathcal{P}_{\tau_2}(f[z]))
\end{array}$$

For example, $\mathcal{P}_{\mathsf{Set}\,\{x:\mathsf{Int} \mid p(x)\}}(s) = \forall z \in s.\, z \in \mathsf{Int} \wedge p(z)$.

Definition 2 (Relativization). *A typed formula is relativized by recursively replacing the type annotations $x : \tau$ by a new hypothesis corresponding to the typing proposition $\mathcal{P}_\tau(x)$. The relevant transformation is $\forall x : \tau.\, \phi \rightsquigarrow \forall x.\, \mathcal{P}_\tau(x) \Rightarrow \phi$.*

Lemma 1 (Relativization is sound). $\vdash \forall x : \tau.\, \phi$ *implies* $\vdash \forall x.\, \mathcal{P}_\tau(x) \Rightarrow \phi$.

Proof. The proof follows [13] with the addition of the Set and refinement types. □

Now suppose we want to annotate the formula $\forall x, y.\ \cup\{x, y\} = \cup\{y, x\}$. We can safely say that the type of x and y should be $\mathsf{Set}\,\mathtt{t}$, for some atomic type $\mathtt{t}$. Semantically speaking, all values in the untyped universe $\mathcal{D}$ denote sets. And the stratification of sets using the Set constructor supports the key idea that a set must have a different type from its elements.

Definition 3 (Safe types). *A type is said to be* safe *if it is an atomic type $\mathtt{t}_i$, for some i, or if it is $\mathsf{Set}\,\tau_{safe}$, where τ_{safe} is safe.*

Since all values are sets and typing predicates are uninterpreted, safe types cannot introduce any unsoundness to a typed formula.

Lemma 2. *The relativization of the formula $\forall x : \tau_{safe}.\, \phi$ is equisatisfiable with $\forall x.\, \phi$.*

Proof. By the definitions of relativization and typing proposition of Set and atomic types. □

In this paper we are going in the opposite direction, that is, from an unsorted to a many-sorted universe. We will obtain the type information from propositions that appear in the unsorted language in the form of *typing hypotheses*.[5]

Definition 4 (Typing hypothesis). *A* typing hypothesis $\mathcal{H}(x)$ *for variable the* x *is a premise of the form* $x \in e$ *or* $x = e$*, for any expression* e *where* x *is not free in* e*.*

The type information that can be obtained from an untyped formula is almost directly taken from their typing hypotheses and can be captured with precision by refinement types. Suppose we want to annotate the invalid formula $\forall x.\ x + 0 = x$. It is incorrect to say that x is an integer: that would make the formula valid. However, the formula $\forall x.\ x \in Nat \Rightarrow x + 0 = x$ contains a hypothesis from which we can soundly infer the type $\{y : \mathsf{Int} \mid 0 \leq y\}$ for x. With this in mind, we define the typing rules.

3.2 Typing rules

We start by declaring some conventional auxiliary definitions. A *typing context* $\Gamma : \mathcal{V} \cup \mathcal{O} \to \tau$ is a finite partial function from variable and operator symbols to types. Its grammar is $\Gamma ::= x : \tau \mid \Gamma, x : \tau$. A triple $\Gamma \vdash \phi : \tau$ is a *pre-judgement*. It is a (valid) *judgement* if it can be derived from the typing rules. A pair (Γ, τ) is a *typing* of ϕ iff $fv(\phi) \subseteq dom(\Gamma)$ and $\Gamma \vdash \phi : \tau$ is valid. Likewise, the typing of a formula is just Γ. A formula ϕ is *typable* iff it admits a typing. Given an untyped formula $\phi \triangleq \forall x.\ \varphi$ such that $\Gamma \vdash \varphi : \mathsf{Bool}$ is a judgment and $fv(\varphi) \subseteq dom(\Gamma)$, then the corresponding *annotated* (sorted) formula is $\phi' \triangleq \forall x : \Gamma(x).\ \varphi$.

The definition of the typing rules is similar to the standard rules for simple typed λ-calculus. The typing rules introduce many fresh type variables noted $\alpha, \alpha_1, \alpha_2, ...$, etc. during a type derivation. In contrast to type inference in programming languages where type variables are unified throughout the whole derivation to obtain a most general type, here we just want to unify variables when deriving the typing hypotheses. In the rest of the formula, we just check that types are well-formed. The core of the typing rules lies in the definition of four binary relations on types. Equality $\equiv$ and subtyping $<:$ are used to unify type variables. They have their corresponding non-unifiable versions: equality checking $\approx$ and subtype checking $\prec:$. Unless explicitly noted, they are all interpreted in a context Γ, for example, as $\Gamma \vdash \tau_1 \equiv \tau_2$, to bind the free variables the refinement predicates may have.

The equality condition $\tau_1 \equiv \tau_2$ tries to unify both types when one of them is a type variable. The subtyping relation $<:$ is a pre-order on types (i.e., it is reflexive and transitive). For any ground types τ_1 and τ_2, $\tau_1 <: \tau_2$ iff $\forall x.\ \mathcal{P}_{\tau_1}(x) \Rightarrow \mathcal{P}_{\tau_2}(x)$; when at least one of τ_1 and τ_2 is a type variable, the types are unified, as explained

[5] TLA$^+$ was designed with the philosophy that the user should not think in terms of types when she writes the specifications and proofs. In practice, it is customary that the first thing the user does after declaring the variables in a TLA$^+$ module is to write a *type invariant* for every declared variable. Once proved, this invariant is used as a hypothesis in the other theorems.

Typing rules for first-order formulas and set objects

$$\frac{}{\Gamma \vdash \mathsf{false} : \mathsf{Bool}}\ [\text{T-FALSE}] \qquad \frac{\Gamma \vdash \phi_1 : \mathsf{Bool} \quad \Gamma \vdash \phi_2 : \mathsf{Bool}}{\Gamma \vdash \phi_1 \Rightarrow \phi_2 : \mathsf{Bool}}\ [\text{T-IMPLIES}]$$

$$\frac{\Gamma, x : \alpha \vdash \phi : \mathsf{Bool}}{\Gamma \vdash \forall x.\ \phi : \mathsf{Bool}}\ [\text{T-QUANT}] \qquad \frac{\Gamma, x : \tau \vdash \phi : \mathsf{Bool}}{\Gamma \vdash \forall x : \tau.\ \phi : \mathsf{Bool}}\ [\text{T-CHECK}]$$

$$\frac{\Gamma(x) \equiv \alpha}{\Gamma \vdash x : \alpha}\ [\text{T-VAR}] \qquad \frac{\Gamma(w) \equiv \alpha_1 \to \alpha_2 \quad \Gamma \vdash e : \alpha_1}{\Gamma \vdash w(e) : \alpha_2}\ [\text{T-OP}]$$

$$\frac{\Gamma \vdash s : \mathsf{Set}\,\alpha \quad \Gamma, x : \alpha \vdash \phi : \mathsf{Bool} \quad x \notin fv(s)}{\Gamma \vdash \{x \in s : \phi\} : \mathsf{Set}\,\{x : \alpha \mid \phi\}}\ [\text{T-SETCOMP}]$$

$$\frac{}{\Gamma \vdash \{\} : \mathsf{Set}\,\varnothing_\alpha}\ [\text{T-EMPTY}] \qquad \frac{\Gamma \vdash e_1 : \alpha_1 \quad \Gamma \vdash e_2 : \alpha_2}{\Gamma \vdash \{e_1, e_2\} : \mathsf{Set}\,(\alpha_1 \uplus \alpha_2)}\ [\text{T-PAIR}]$$

$$\frac{\Gamma \vdash s : \mathsf{Set}\,\alpha}{\Gamma \vdash \mathbb{P}s : \mathsf{Set}\,\mathsf{Set}\,\alpha}\ [\text{T-POWER}] \qquad \frac{\Gamma \vdash s : \mathsf{Set}\,\mathsf{Set}\,\alpha}{\Gamma \vdash \cup s : \mathsf{Set}\,\alpha}\ [\text{T-UNION}]$$

$$\frac{\Gamma \vdash e_1 : \alpha_1 \quad \Gamma \vdash \alpha_1 \prec: \alpha_3 \quad \Gamma \vdash e_2 : \alpha_2 \quad \Gamma \vdash \alpha_2 \prec: \alpha_3}{\Gamma \vdash e_1 = e_2 : \mathsf{Bool}}\ [\text{T-EQ}] \qquad \frac{\Gamma \vdash e_1 : \alpha_1 \quad \Gamma \vdash e_2 : \mathsf{Set}\,\alpha_2 \quad \Gamma \vdash \alpha_1 \prec: \alpha_2}{\Gamma \vdash e_1 \in e_2 : \mathsf{Bool}}\ [\text{T-MEM}]$$

$$\frac{\Gamma \vdash e : \alpha \quad \Gamma, x : \alpha \vdash \phi : \mathsf{Bool} \quad x \notin fv(e)}{\Gamma \vdash \forall x.\ x = e \Rightarrow \phi : \mathsf{Bool}}\ [\text{TH-EQ}] \qquad \frac{\Gamma \vdash e : \mathsf{Set}\,\alpha \quad \Gamma, x : \alpha \vdash \phi : \mathsf{Bool} \quad x \notin fv(e)}{\Gamma \vdash \forall x.\ x \in e \Rightarrow \phi : \mathsf{Bool}}\ [\text{TH-MEM}]$$

Typing rules for function and arithmetic expressions

$$\frac{\Gamma \vdash f : \alpha_1 \quad \Gamma \vdash \alpha_1 \approx (x : \alpha_3) \to \alpha_4 \quad \Gamma \vdash e : \alpha_2 \quad \Gamma \vdash \alpha_2 \prec: \alpha_3}{\Gamma \vdash f[e] : [x \mapsto e] \cdot \alpha_4}\ [\text{T-APP}] \qquad \frac{\Gamma \vdash f : \alpha_1 \quad \Gamma \vdash \alpha_1 \approx (x : \alpha_2) \to \alpha_3}{\Gamma \vdash \mathsf{dom}\, f : \mathsf{Set}\,\alpha_2}\ [\text{T-DOM}]$$

$$\frac{\Gamma \vdash s : \mathsf{Set}\,\alpha_1 \quad \Gamma, x : \alpha_1 \vdash e : \alpha_2}{\Gamma \vdash \lambda x \in s.\ e : (x : \alpha_1) \to \alpha_2}\ [\text{T-FUN}] \qquad \frac{\Gamma \vdash s : \mathsf{Set}\,\alpha_1 \quad \Gamma \vdash t : \mathsf{Set}\,\alpha_2}{\Gamma \vdash [s \to t] : \mathsf{Set}\,(\alpha_1 \to \alpha_2)}\ [\text{T-FUNSET}]$$

$$\frac{}{\Gamma \vdash \mathit{Int} : \mathsf{Set}\,\mathsf{Int}}\ [\text{T-INT}] \qquad \frac{\Gamma \vdash e_i : \alpha_i \quad \Gamma \vdash \alpha_i \prec: \mathsf{Int} \quad i \in \{1, 2\}}{\Gamma \vdash e_1 + e_2 : \{x : \mathsf{Int} \mid x = e_1 + e_2\}}\ [\text{T-PLUS}]$$

$$\frac{n \in \{0, 1, 2, ...\}}{\Gamma \vdash n : \{x : \mathsf{Int} \mid x = n\}}\ [\text{T-NUM}] \qquad \frac{\Gamma \vdash e_i : \alpha_i \quad \Gamma \vdash \alpha_i \prec: \mathsf{Int} \quad i \in \{1, 2\}}{\Gamma \vdash e_1 < e_2 : \mathsf{Bool}}\ [\text{T-LESS}]$$

Rules for $\lessdot:$ (that is, $<:$ or $\prec:$) and $\approx$

$$\frac{\Gamma \vdash e : \tau_1 \quad \Gamma \vdash \tau_1 \lessdot: \tau_2}{\Gamma \vdash e : \tau_2}\ [\text{T-SUB}] \qquad \frac{\Gamma \vdash \alpha_1 \equiv \tau_1 \quad \Gamma \vdash \alpha_2 \equiv \tau_2}{\Gamma \vdash (x : \tau_1) \to \tau_2 \approx (x : \alpha_1) \to \alpha_2}\ [\text{MATCH-ARROW}]$$

$$\frac{\Gamma, x : \tau \vdash \phi_1 \Leftrightarrow \phi_2}{\Gamma \vdash \{x : \tau \mid \phi_1\} \equiv \{x : \tau \mid \phi_2\}}\ [\text{EQ-REF}] \qquad \frac{\Gamma \vdash \tau_1 \equiv \tau_1' \quad \Gamma \vdash \tau_2 \equiv \tau_2'}{\Gamma \vdash (x : \tau_1) \to \tau_2 \equiv (x : \tau_1') \to \tau_2'}\ [\text{EQ-ARROW}] \qquad \frac{\Gamma \vdash \tau_1 \equiv \tau_2}{\Gamma \vdash \mathsf{Set}\,\tau_1 \equiv \mathsf{Set}\,\tau_2}\ [\text{EQ-SET}]$$

$$\frac{\Gamma, x : \tau \vdash \phi_1 \Rightarrow \phi_2}{\Gamma \vdash \{x : \tau \mid \phi_1\} \lessdot: \{x : \tau \mid \phi_2\}}\ [\text{SUB-REF}] \qquad \frac{\Gamma \vdash \tau_1' \lessdot: \tau_1 \quad \Gamma, x : \tau_1' \vdash \tau_2 \lessdot: \tau_2'}{\Gamma \vdash (x : \tau_1) \to \tau_2 \lessdot: (x : \tau_1') \to \tau_2'}\ [\text{SUB-ARROW}] \qquad \frac{\Gamma \vdash \tau_1 \lessdot: \tau_2}{\Gamma \vdash \mathsf{Set}\,\tau_1 \lessdot: \mathsf{Set}\,\tau_2}\ [\text{SUB-SET}]$$

Fig. 2. Typing and subtyping rules.

later by the rules of constraint solving. The condition $\tau_1 \prec: \tau_2$ is valid iff both types are ground types and $\tau_1 <: \tau_2$. That is, it checks that τ_1 is a subtype of τ_2, without unifying type variables. We use the symbol $\lessdot:$ as a shorthand for $<:$ and $\prec:$. The rules EQ-REF and SUB-REF yield type verification conditions on first-order formulas that have to be proved correct to satisfy the type property. Therefore, the verification of these conditions is an undecidable problem [17]. Well-formedness conditions on types reduce basically to check the type conditions.

The typing rules are given in Figure 2. As expected, once a formula has been Boolified (cf. Section 2), the rules for false and $\Rightarrow$ are trivial. Rule T-QUANT evaluates the body of $\forall x.\ \phi$ by adding x to the context with a fresh type variable α. We obtain the typing hypotheses by decomposing the assumptions present in a formula by elementary heuristics. The rules TH-EQ and TH-MEM, which are applied with higher priority than rule T-QUANT, encapsulate this requirement in a simplified way. However, the information provided by the typing hypotheses may not be completely captured by merely syntactic analysis. For example, the typing proposition $\mathcal{P}_{\mathsf{Set\,Int}}(s)$ is equal to $\forall z \in s.\ z \in \mathit{Int}$, but the typing hypothesis may appear, for instance, as the equivalent formula $s \in \mathbb{P}\mathit{Int}$. The sub-expressions $x \in s$ in the rules T-SETCOMP and T-FUN are typing hypotheses and are therefore treated as such.

The precise type information of refinement types imposes a weak form of type equality (rule T-EQ). If we require the types of the arguments to be exactly equal, we would be ruling out many typable expressions. Instead, the rule requires them to a have common super-type. Suppose we want to type the expression $3 = 4$. It is false, but still typable because the types $\{x : \mathsf{Int} \mid x = 3\}$ and $\{x : \mathsf{Int} \mid x = 4\}$, which have the same base type Int, are both subtypes of $\{x : \mathsf{Int} \mid x = 3 \vee x = 4\}$.

Functions are contravariant on their arguments while they are covariant on their result (rule SUB-ARROW). This has the effect of shrinking their domain while expanding their codomain. To extract the domain from a function type, as needed by rules T-APP and T-DOM, we use the condition $\tau_1 \approx \tau_2$ as a kind of pattern-matching for functions (MATCH-ARROW). When τ is a function type and α_1 and α_2 fresh variables, $\tau \approx (x : \alpha_1) \to \alpha_2$ obtains the domain of τ in α_1 and the codomain in α_2. Function applications (T-APP) have type $[x \mapsto e] \cdot \alpha_4$: it is the type α_4 of the function's codomain, to which it is applied a substitution of the variable x by expression e. The substitution has to be delayed until it is applied to a refinement type, when we can simplify it as:

$$[x \mapsto e] \cdot \{x' : \tau \mid \phi\} \quad \longrightarrow \quad \{x' : \tau \mid \phi[x \leftarrow e]\}$$

Literal integers and the set of integers have a constant type (T-NUM and T-INT). Rules T-PLUS and T-LESS require that their arguments to be integers with the condition $e_i \prec: \mathsf{Int}$. The rule for $x - y$ is similar to the rule T-PLUS.

Finally, to type check an annotated formula, we use the same type system, except that the typing rules TH-MEM, TH-EQ and T-QUANT for quantifiers are no longer needed; they are replaced by the rule T-CHECK. This means that during type checking there are no derivations from typing hypotheses, and type annotations in quantifiers are passed directly to the body's context.

3.3 Soundness

Type annotations, as well as the typing hypotheses, restrict the domain of evaluation of the quantified variables. Suppose the formula ϕ is not valid. Then there exists some valuation in the universe $\mathcal{D}$ which makes the formula false. Still, there may exist some other valuation in $\mathcal{D}$ that makes ϕ true. Let us call A the set of all valuations that make ϕ true. We want to show that the type system does not generate annotations for ϕ, resulting in ϕ', such that those annotations restrict or confine the domain of evaluation of the variables to the set A which would make ϕ' valid.

For example, consider $\forall x.\, x < x+1$ which is false in some valuations of x, namely when $x \notin Int$. However, if we annotate x incorrectly as an integer, $\forall x : \mathsf{Int}.\, x < x+1$ would become valid, because x would be evaluated precisely in those values that make $x < x+1$ true. In essence, we need to prove that type assignments only follow from typing hypotheses.

Theorem 1 (Soundness). *If $x : \tau$ is a typing of ϕ, then $\vdash \forall x.\, \phi$ iff $\vdash \forall x : \tau.\, \phi$.*

Proof. $\Rightarrow$) If ϕ is true in all models of the untyped universe, then in a sorted universe that restricts the domain of interpretation, ϕ will also be trivially true.

$\Leftarrow$) Assuming $\vdash \forall x : \tau.\, \phi$ (named A_1) we want to prove $\vdash \forall x.\, \phi$.

PROOF. We know that:

$\langle 1 \rangle 1$. $x : \tau \vdash \phi : \mathsf{Bool}$ is valid (i.e. there is a type derivation), by hypothesis.

$\langle 1 \rangle 2$. $\vdash \forall x.\, \mathcal{P}_\tau(x) \Rightarrow \phi$ (named A_2), by assumption A_1 and Lemma 1.

We need to show that $\mathcal{P}_\tau(x)$, derived from $x : \tau$, does not constraint the domain of evaluation of x in ϕ.

$\langle 1 \rangle 3$. Suffices to prove that from $\vdash A_2$ we can prove $\vdash \forall x.\, \phi$, by step $\langle 1 \rangle 2$.

We proceed by a case analysis on the shape of ϕ.

$\langle 1 \rangle 4$. CASE 1. If there is no typing hypothesis for the variable x in ϕ, then $\vdash \forall x.\, \phi$.

PROOF.

$\langle 2 \rangle 1$. The type derivation on ϕ yields the judgment $x : \alpha_x \vdash \phi : \mathsf{Bool}$, by step $\langle 1 \rangle 1$. Type variable α_x is fresh and after unification will be equal to τ. The first applied rule is T-QUANT, the only possible one, since there are no typing hypotheses.

$\langle 2 \rangle 2$. The type α_x can only be promoted to a safe type τ.

PROOF. The TH (typing hypothesis) rules, where unification of type variables happens, do not apply, meaning that α_x cannot be unified with any non-safe type such as Bool, Int or functions. The only applicable rules that may promote α_x are the rules T-MEM, T-SETCOMP, T-PAIR, T-POWER or T-UNION, but these result in a safe Set type. For example, rule T-PLUS requires establishing $\alpha_x \prec: \mathsf{Int}$, which is impossible.

$\langle 2 \rangle 3$. Finally, since τ is safe, it does not compromise the validity of A_2 when $x : \tau$ it is relativized to $\mathcal{P}_\tau(x)$, by Lemma 2.

$\langle 1 \rangle 5$. CASE 2. If ϕ is of the form $\mathcal{H}(x) \Rightarrow \phi_1$, then $\vdash \forall x.\, \mathcal{H}(x) \Rightarrow \phi_1$.

PROOF.

$\langle 2 \rangle 1$. Suffices to prove that $\mathcal{H}(x) \Rightarrow \mathcal{P}_\tau(x)$.

$\langle 2 \rangle 2$. Suppose that $\mathcal{H}(x)$ is of the form $x \in s$. The first rule applied in the type derivation is necessarily TH-MEM, yielding

$$\vdash s : \mathsf{Set}\,\alpha_x \;\; (1) \qquad \text{and} \qquad x : \alpha_x \vdash \phi_1 : \mathsf{Bool} \;\; (2)$$

Here, we see that the fresh type variable α_x is the same in both sides of the derivation, which results in the unification of the types of x and s. The TH rules are the only ones that share type variables in their different premises.

We apply induction on $fv(\mathcal{H}(x))$. For simplicity, we consider that $\mathcal{H}(x)$ does not include quantified formulas.

$\langle 3 \rangle 1$. (Base case) There are no free variables, meaning that the type of x does not depend on the type of any other variable. Therefore, it is trivially a constant type or an atomic type t. For instance, if s is Int, the goal is to show that $x \in Int \Rightarrow \mathcal{P}_{\alpha_x}(x)$. So α_x is unified with Int and $\mathcal{P}_{\mathsf{Int}}(x) = x \in Int = \mathcal{H}(x)$.

$\langle 3 \rangle 2$. (Inductive step) We proceed by a case analysis on the shape of s, which has to be necessarily a set, otherwise it would not match with $\mathsf{Set}\,\alpha_x$ in (1).

$\langle 4 \rangle 1$. CASE $s \triangleq \mathbb{P}t$. The goal is to show that $x \in \mathbb{P}t \Rightarrow \mathcal{P}_{\alpha_x}(x)$. Given that $t : \alpha_t$, then α_x is unified with $\mathsf{Set}\,\alpha_t$. Then $\mathcal{P}_{\mathsf{Set}\,\alpha_t}(x) = \forall z \in x.\ \mathcal{P}_{\alpha_t}(z)$, by the inductive hypothesis $z \in t \Rightarrow \mathcal{P}_{\alpha_t}(z)$.

$\langle 4 \rangle 2$. The other cases are proved in a similar way.

$\langle 2 \rangle 3$. The case where $\mathcal{H}(x)$ is of the form $x = e$ is similar to the step $\langle 2 \rangle 2$.

$\langle 2 \rangle 4$. QED, by $\langle 2 \rangle 1$, $\langle 2 \rangle 2$ and $\langle 2 \rangle 3$.

$\langle 1 \rangle 6$. QED, by steps $\langle 1 \rangle 3$, $\langle 1 \rangle 4$ and $\langle 1 \rangle 5$. □

4 Type Inference Algorithm

The type inference algorithm takes a formula ϕ and returns a type assignment σ, that is, a function from type variables to types. The algorithm consists of a constraint generation phase followed by constraint solving.

Since the constraint-based algorithm is independent of the chosen type system we can adapt one originally introduced for a variant of ML by Knowles and Flanagan [10]. The main difference is in the constraint language, where we use two additional kinds of type checking conditions instead of only two for equality and subtyping. The constraint language grammar is defined following the notation of [16].

$$c ::= \tau \equiv \tau \mid \tau <: \tau \mid \tau \approx \tau \mid \tau \prec: \tau \mid \top \mid \bot \mid c \wedge c \mid \exists \vec{\alpha}.\, c \mid [x \mapsto e] \cdot c$$

In addition to the type constraints, there are the true and false constraints. Conjunction of constraints and existential quantification of type variables permit to replicate the structure of a type derivation in a single constraint formula. Delayed substitutions $[x \mapsto e] \cdot c$ replace variable x by expression e in constraint c.

A constraint c is *satisfiable*, noted $\sigma \vdash c$, iff there exists a ground assignment σ that satisfies c. Constraint judgements can be interpreted by the following rules, where $\sigma, \alpha \mapsto \mathsf{t}$ is function σ updated with a new assignement for α and t is fresh atomic type:

$$\frac{}{\sigma \vdash \top} \qquad \frac{\sigma\tau_1 \not\downarrow \sigma\tau_2}{\sigma \vdash \tau_1 \not\downarrow \tau_2} \;\; (\not\downarrow \in \{\equiv, \simeq, <:, \prec:\}) \qquad \frac{\sigma\, c_1 \quad \sigma\, c_2}{\sigma \vdash c_1 \wedge c_2} \qquad \frac{\sigma, \alpha \mapsto \mathsf{t} \vdash c}{\sigma \vdash \exists \alpha.\, c}$$

4.1 Constraint generation

To a pre-judgement $\Gamma \vdash e : \tau$, where $fv(e) \subseteq dom(\Gamma)$, we associate a constraint $\langle\!\langle \Gamma \vdash e : \tau \rangle\!\rangle$. Constraint generation (CG) rules are essentially derived from their corresponding typing rules, with subsumption (rule T-SUB) distributed all through to make the rules syntax-directed. CG rules take as arguments an environment Γ, an expression e and a type variable τ. They are recursively defined on e. The resulting constraint has a linear size with respect to the size of the original formula. As an example, we show the CG rule obtained from the rule T-SETCOMP:

$$\begin{aligned}\langle\!\langle \Gamma \vdash \{x \in s : \phi\} : \alpha_r \rangle\!\rangle \triangleq \exists \alpha_1 \alpha_2.\ & \wedge \langle\!\langle \Gamma \vdash s : \mathsf{Set}\,\alpha_1 \rangle\!\rangle \\ & \wedge \langle\!\langle \Gamma, x : \alpha_2 \vdash \phi : \mathsf{Bool} \rangle\!\rangle \\ & \wedge \Gamma \vdash \alpha_2 \prec: \alpha_1 \\ & \wedge \Gamma \vdash \alpha_r \equiv \mathsf{Set}\,\{x : \alpha_1 \mid \phi\}\end{aligned}$$

Note that: (i) every free type variable that appears in the typing rule are existentially bounded by fresh type variables α_1 and α_2, (ii) the expected type for the expression in the second argument is unified to the type variable α_r passed as the third argument, and (iii) the subsumption rule is implicitly applied to the sub-formula $x \in s$.

The following theorem asserts that the soundness and completeness of the generated constraints, grounded by a type assignment σ.

Theorem 2 (CG soundness and completeness). $\sigma \vdash \langle\!\langle \Gamma \vdash \phi : \tau \rangle\!\rangle$ *iff* $\sigma\Gamma \vdash \phi : \sigma\tau$.

Proof (idea). By induction on ϕ, using the typing rules, the CG definitions and the interpretation of constraints. For details, see [17]. □

4.2 Constraint Solving

Constraint-based type inference for systems with subtyping is an extensive research topic. Pottier [16] and Odersky et al. [15] have developed Hindley-Milner systems parameterized by a subtyping constraint system. Broadly speaking, we specify a constraint solving algorithm following [9] as a non-deterministic system of constraint rewriting rules and first-order unification rules for subtyping constraints. The algorithm proceeds in one main step, that is repeated once, consisting of solving equality and subtyping constraints. Once the first execution is finished, the final typing we were searching for is Γ, but there are still some residual subtype checking constraints of the form $\tau_1 \prec: \tau_2$ to prove. The second step is to check that these constraints are satisfied, by converting them to the form $\tau_1 <: \tau_2$ and solving them by executing the main step again. If the remaining constraint is $\top$, the algorithm finishes successfully.

To solve the equality and subtyping constraints we proceed as follows. Given a context Γ and a constraint c, we apply the rules 3.1, 3.2, 3.3 and MATCH-ARROW plus the following rules to eliminate the type variables introduced during constraint generation. Note that rule 4.2 has to be carefully applied to avoid recursive substitutions.

$$(\exists \alpha.\ c_1) \wedge c_2 \longrightarrow \exists \alpha.\ (c_1 \wedge c_2) \qquad \text{if } \alpha \notin fv(C_2) \tag{4.1}$$

$$\exists \alpha.\ (\Gamma \vdash \alpha \equiv \tau \wedge c) \longrightarrow c[\tau \leftarrow \alpha] \qquad \text{if } \alpha \text{ does not occur in } \tau \tag{4.2}$$

Subtype constraints $\Gamma \vdash \tau_1 <: \tau_2$ are solved by non-deterministically applying simplification rules SUB-REF, SUB-ARROW, and SUB-SET, or the unification rules:

$$\begin{array}{rcll}
\Gamma \vdash \mathsf{Set}\,\tau <: \sigma \cdot \alpha & \rightsquigarrow & \{\alpha \Rightarrow \mathsf{Set}\,\tau\} & \\
\Gamma \vdash (x : \tau_1) \to \tau_2 <: \sigma \cdot \alpha & \rightsquigarrow & \{\alpha \Rightarrow (x : \alpha_1) \to \alpha_2\} & (\alpha_1, \alpha_2 \text{ fresh variables}) \\
\Gamma \vdash \{x : \tau \mid \phi\} <: \sigma \cdot \alpha & \rightsquigarrow & \{\alpha \Rightarrow \{x : \tau \mid \gamma\}\} & (\gamma \text{ fresh placeholder}) \\
\Gamma \vdash \{x : \tau \mid \phi_1\} <: \{x' : \alpha \mid \phi_2\} & \rightsquigarrow & \{\alpha \Rightarrow \{x : \tau \mid \phi_1\}\} &
\end{array}$$

These four unification rules have their symmetric counterparts. They return a substitution $\{\alpha \Rightarrow \tau\}$ of a variable α by another type τ, which are immediately applied to Γ and c. Any other pair combination of set, function or refinement types will make the algorithm abort with a type error. The algorithm terminates when no rule can be applied. At this point, only subtype constraints $\alpha_1 <: \alpha_2$ between type variables remain in c. The type variables α_1 and α_2 can be set to a concrete ground type `t`, making the constraint valid by reflexivity. Placeholder symbols are introduced to defer the reconstruction of refinement predicates.

Solving placeholders. The final step in type inference algorithm is to find formulas to replace the placeholders while satisfying the typing conditions. The placeholders appear in conditions of the form $\Gamma \vdash \gamma \Rightarrow \phi$, $\Gamma \vdash \gamma_1 \Rightarrow \gamma_2$ or $\Gamma \vdash \phi \Rightarrow \gamma$. Our algorithm to calculate concrete refinement predicates is almost entirely based on a similar one developed in [10], which, in turn, is based on the intuition that implications can be analyzed as dataflow graphs.

5 Experimental Results

We have implemented a prototype of the type inference algorithm in TLAPS. In particular, the following table shows results for two case studies. They correspond to the invariant proofs of the N-process Peterson and Bakery algorithms for mutual exclusion, whose data structures are represented by functions ranging over the processes and they contain some basic arithmetic.

For each benchmark, we record the size of the proof, i.e. the number of non-trivial proof obligations generated by the proof manager, and the time in seconds required to verify those proofs on a standard laptop. The proof size corresponds to the number of proof obligations that are passed to the backend prover, which is proportional to the number of interactive steps and therefore represents the user effort for making TLAPS check the proof. We compare these figures for the SMT backend using the previous elementary type inference algorithm described in [14], and then for the SMT backend equipped with the new type system with refinement types. The results for the new type system includes three extra columns corresponding to the number of derived type verification conditions (non-trivial vs. total), total time in seconds to perform the type inference (including proving the type conditions), and number of initially generated constraints. The total time for the second system is the sum of the times required to do type inference and the time to actually prove the SMT encoding of the proof obligation. In all cases, the SMT solver used was CVC4 [3].

	Simple Types		Refinement Types				
	size	time	size	time	tvc	type-inf	const
Peterson	3	0.40	3	0.30	0/474	0.33	937
Bakery	15	9.52	3	1.51	6/1622	4.15	3317

The second case study, which is a significantly bigger specification than the first one, takes slightly less time than the previous backend, whereas the overall time taken is slightly longer for the Peterson case study. The size of the proof, i.e. the number of human interactions, is considerably reduced for the second case. The current prototypical implementation of the constraint solving algorithm may benefit from optimizations (see [16]) in order to speed up type inference.

In both examples, all non-trivial verification conditions were discharged almost instantly by the SMT solver. Consequently, no dynamic domain checkings were needed in the SMT-LIB encoding.

6 Conclusions

Beyond the recurring debates about using typed versus untyped languages for formalizing mathematics or software systems [12], we can observe that types, regarded just as a classification of the elements of a language, arise quite naturally in untyped set theory. In this paper, motivated by the use of powerful automatic provers for multi-sorted first-order logic, we have defined a sophisticated type system for a fragment of the TLA$^+$ specification language that captures with precision the values and semantics of sets and functions using refinement and dependent types. When type inference succeeds, we obtain type annotations on top of an untyped specification language, getting the best of both the typed and untyped approaches.

Inevitably, the resulting type system constrains the set of accepted TLA$^+$ expressions. Occasionally useful expressions that are not typable by the type system are, for example, enumerated sets whose elements are of different types. As we mentioned in the introduction, formulas for which type inference fails will still be translated according to the "untyped" encoding, and may thus be proved by the SMT solver. One advantage of doing type inference with constraints is that we can know exactly what part of the formula cannot be typed and can therefore restrict the use of the untyped encoding to these parts and produce useful type checking warnings and error messages [8].

Our experience so far with the implementation of this approach in TLAPS has been quite positive: types are successfully inferred for the vast majority of proof obligations that we have seen in practice. Since the new type system is a refinement of the previous one, it never fails when the old one succeeded, and it has been able to increase the number of proof obligations that the SMT backend can handle without human interaction. The improvements are particularly noticeable in specifications that contain a significant number of function applications, which are used quite frequently in TLA$^+$ specifications.

The type system is easily extended to accommodate TLA$^+$ constructs that we have not considered in this paper, such as tuples and records. Support for the CHOOSE operator (Hilbert's choice) is more challenging. It would be interesting to study the applicability of our type system to proofs of mathematical theorems in ZF set theory.

References

1. J.-R. Abrial. *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010.
2. D. Aspinall and A. B. Compagnoni. Subtyping dependent types. *Theor. Comput. Sci.*, 266(1-2):273–309, 2001.
3. C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. Cvc4. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 171–177, Berlin, Heidelberg, 2011. Springer-Verlag.
4. C. Barrett, A. Stump, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010.
5. D. Déharbe. Integration of SMT-solvers in B and Event-B development environments. *Sci. Comput. Program.*, 78(3):310–326, 2013.
6. G. Dowek. Collections, sets and types. *Mathematical. Structures in Comp. Sci.*, 9(1):109–123, Feb. 1999.
7. T. Freeman and F. Pfenning. Refinement types for ML. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, PLDI '91, pages 268–277, New York, NY, USA, 1991. ACM.
8. B. Heeren, J. Hage, and D. Swierstra. Generalizing Hindley-Milner type inference algorithms. Technical report, 2002.
9. J.-P. Jouannaud and C. Kirchner. Solving equations in abstract algebras: A rule-based survey of unification. In *Computational Logic - Essays in Honor of Alan Robinson*, pages 257–321, 1991.
10. K. Knowles and C. Flanagan. Type reconstruction for general refinement types. In *Proceedings of the 16th European conference on Programming*, ESOP'07, pages 505–519, Berlin, Heidelberg, 2007. Springer-Verlag.
11. L. Lamport. *Specifying Systems: The TLA$^+$ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, Boston, Mass., 2002.
12. L. Lamport and L. C. Paulson. Should your specification language be typed? *ACM Trans. Program. Lang. Syst.*, 21(3):502–526, May 1999.
13. M. Manzano. *Extensions of First-Order Logic*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2nd edition, 2005.
14. S. Merz and H. Vanzetto. Harnessing SMT Solvers for TLA$^+$ Proofs. *ECEASST*, 53, 2012.
15. M. Odersky, M. Sulzmann, and M. Wehr. Type inference with constrained types. In *Fourth International Workshop on Foundations of Object-Oriented Programming (FOOL)*, 1997.
16. F. Pottier. Simplifying subtyping constraints. In *In Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pages 122–133. ACM Press, 1996.
17. F. Pottier and D. Rémy. The essence of ML type inference. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.
18. J. Rushby, S. Owre, and N. Shankar. Subtypes for Specifications: Predicate Subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709–720, sep 1998.
19. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 1992.
20. H. Xi and F. Pfenning. Dependent types in practical programming. In A. W. Appel and A. Aiken, editors, *POPL*, pages 214–227. ACM, 1999.