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^+ \cite{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 \cite{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 \cite{19} or B \cite{1} use typed variants of set theory that correspond naturally to multi-sorted first-order logic \cite{5}.

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

\[
\forall m, n : \text{Int}. \, \text{plus} (\text{int2u}(m), \text{int2u}(n)) = \text{int2u}(m + n).
\]
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 ∀x. x ∈ Int ⇒ x + 0 = x in SMT-LIB

With this representation, the SMT backend will be unable to prove the TLA⁺ formula ∀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} + 0 is syntactically correct, but its value is unspecified. However, the TLA⁺ formula ∀x. x ∈ Int ⇒ 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 ∈ Int in the TLA⁺ formula gives rise to the subformula ∃n : Int. x = int2u(n). If we could detect appropriate type information from the original TLA⁺ formula, we could simply translate it to ∀x : 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

∀f ∈ [{1, 2, 3} → 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 ∈ 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] ∪ { } = f[x], are actually valid irrespectively of whether x ∈ dom f holds or not. This observation motivates the use of a more expressive type system. Using refinement types [17,20], the type of dom f is \{x : Int | x = 1 ∨ x = 2 ∨ x = 3\}. During type inference, the system will try to prove that x = 0 ⇒ x ∈ dom f, and this will fail, hence the translation

---

1 In TLA⁺, [S → 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].