Type Inference for TLA$^+$ in Apalache *

Jure Kukovec[1] and Igor Konnov[2]

[1] TU Wien, Vienna, Austria
Email: jkukovec@forsyte.at
[2]Informal Systems, Vienna, Austria
Email: igor@informal.systems

Abstract

TLA$^+$ is a general specification language introduced by Leslie Lamport to specify all kinds of computer systems: from sequential algorithms to concurrent and distributed systems. Recently, this language has been gaining popularity in distributed systems, as an alternative to documenting protocols in pseudo-code. Although TLA$^+$ was not designed with automatic analysis tools in mind, the TLA$^+$ ecosystem includes an interactive proof system (TLAPS), an explicit model checker (TLC), and two symbolic model checkers (ProB and Apalache).

The language of TLA$^+$ is an untyped logic. The analysis tools assume a type system, e.g., to encode verification queries in SMT, and do tool-specific type inference for specific fragments of the language. In this paper, we introduce an approach to type inference for TLA$^+$ that does not pose tool-specific language restrictions. Our type system consolidates the type systems of the above tools, which differ in details. We reduce the type inference problem to SMT satisfiability in quantifier-free uninterpreted first-order logic. We have conducted experiments on 36 TLA$^+$ specifications from the TLA$^+$ examples repository. The experiments showed that 30 specifications were type correct, 3 specifications had actual type errors, and only 3 specifications were using the features of the untyped logic.

1 Overview

Since it is hard to develop an automatic analysis tool for a completely untyped language, the existing tools introduce a type system and run some form of type inference or type checking. TLC checks type compatibility when computing system states, e.g., it rejects a set of integers and Booleans $\{2020, \text{TRUE}\}$. TLAPS infers types of the verification conditions that are sent to an SMT solver [3, 4, 5]. ([5] also introduced a purely untyped encoding.) TLA2B implements type inference similar to the one in TLAPS, whereas Apalache [1, 2] performs a simple top-down type propagation and falls back to user annotations, when type inference fails.

In this talk, we discuss three major points. First, we discuss a type system τ^{TLA} that encompasses the type systems of TLAPS, TLA2B, and Apalache. Our type system supports TLA$^+$ operators and thus requires no additional preprocessing.

Second, we present our new type inference technique for τ^{TLA} that translates the type inference problem to SMT constraints (in the theory of quantifier-free uninterpreted first-order formulas).

*This research was supported in part by Interchain Foundation (Switzerland) and the Vienna Science and Technology Fund (WWTF) through project APALACHE (ICT15-103).

This technique is based on operator applications — we annotate each built-in TLA$^+$ operator (CHOOSE , $\cup$, CASE , etc.) with a schema, that is, the most general polymorphic type of the operator. Then, every instance of operator application generates SMT constraints, which encode the compatibility of the concrete argument passed to the operator with the type(s) permitted by the operator schema. From an SMT model, if it exists, we then recover one valid monotype for each TLA$^+$ expression in the specification.

Third, we demonstrate the new type inference tool for TLA$^+$ and the experiments on 36 specifications from the repository of TLA$^+$ examples [6]. Out of 36 benchmarks, only 6 specifications were not well-typed in τ^{TLA}. Three of them contained actual errors that were flagged by the type checker. The three remaining specifications used the untyped semantics. However, we easily made all three specifications well-typed by changing 2-3 lines of code. Importantly, TLA$^+$ specifications rarely have more than 2,000 lines of code. Hence, the use of SMT does not usually impact the tool performance.

Our Type System. We extend the type system introduced by [3] as follows:

$$\begin{aligned}
\tau &::= \alpha \mid \mathsf{Bool} \mid \mathsf{Int} \mid \mathsf{Str} \mid \tau \to \tau \mid \mathsf{Set}(\tau) \mid \mathsf{Seq}(\tau) \mid \\
&\quad \langle \tau, \ldots, \tau \rangle \mid \langle i_1 \mapsto \tau, \ldots, i_k \mapsto \tau \rangle \mid [h_1 \mapsto \tau, \ldots, h_k \mapsto \tau] \mid \langle \tau, \ldots, \tau \rangle \Rightarrow \tau \\
s &::= \forall \vec{\alpha} \,.\, \tau \mid s \sqcup s
\end{aligned}$$

Types like Int, $\mathsf{Set}(\cdot)$, and $\tau_1 \to \tau_2$ represent integers, sets, and TLA$^+$ functions, respectively. Further, we have tuples $\langle \tau_1, \ldots, \tau_k \rangle$, sparse tuples $\langle i_1 \mapsto \tau_1, \ldots, i_k \mapsto \tau_k \rangle$, records $[h_1 \mapsto \tau, \ldots, h_k \mapsto \tau]$, and operators $\langle \tau, \ldots, \tau \rangle \Rightarrow \tau$. There is no special syntax for sparse tuples in TLA$^+$, but we need them to talk about operator signatures. A type $\forall \vec{\alpha} \,.\, \tau$ is considered polymorphic and is used to describe, for example, the type of the built-in set constructor operator $\{\cdot\}$, which may be applied to any type of argument. The role of schema types derived from s is to capture types of overloaded built-in operators, such as, for example, the EXCEPT operator, which is applicable to functions, sequences and records, even tough those types are mutually incompatible.

References

[1] Apalache model checker. Informal Systems, 2020. `https://github.com/informalsystems/apalache`. Last accessed on August 31, 2020.

[2] Igor Konnov, Jure Kukovec, and Thanh-Hai Tran. TLA+ model checking made symbolic. *Proc. ACM Program. Lang.*, 3(OOPSLA):123:1–123:30, 2019.

[3] Stephan Merz and Hernán Vanzetto. Automatic verification of TLA$^+$ proof obligations with SMT solvers. In *LPAR*, volume 7180, pages 289–303. Springer, 2012.

[4] Stephan Merz and Hernán Vanzetto. Refinement types for TLA$^+$. In *NASA Formal Methods Symposium*, pages 143–157. Springer, 2014.

[5] Stephan Merz and Hernán Vanzetto. Encoding tla+ into unsorted and many-sorted first-order logic. *Science of Computer Programming*, 158:3–20, 2018.

[6] A collection of TLA+ specifications of varying complexities, 2020. `https://github.com/tlaplus/Examples`. Last accessed on August 31, 2020.