Towards certification of TLA+ proof obligations with SMT solvers

Stephan Merz, Hernán Vanzetto

Towards certification of TLA$^+$ proof obligations with SMT solvers

Stephan Merz and Hernán Vanzetto*
INRIA Nancy Grand-Est & LORIA
Nancy, France

Abstract

TLA$^+$ is a formal specification language that is based on Zermelo-Fränkel set theory and the Temporal Logic of Actions TLA. The TLA$^+$ proof system TLAPS assists users in deductively verifying safety properties of TLA$^+$ specifications. TLAPS is built around a *proof manager*, which interprets the TLA$^+$ proof language, generates corresponding proof obligations, and passes them to backend verifiers. In this paper we present a new backend for use with SMT solvers that supports elementary set theory, functions, arithmetic, tuples, and records. We introduce a typing discipline for TLA$^+$ proof obligations, which helps us to disambiguate the translation of expressions of (untyped) TLA$^+$, while ensuring its soundness. Our work is a first step towards the certification of proofs generated by proof-producing SMT solvers in Isabelle/TLA$^+$, which is intended to be the only trusted component of TLAPS.

1 Introduction

TLA$^+$ [8] is a language for specifying and verifying concurrent and distributed algorithms and systems. It is based on a variant of Zermelo-Fränkel set theory for specifying the data structures, and on the Temporal Logic of Actions TLA for describing the dynamic system behavior. Recently, a first version of the TLA$^+$ proof system TLAPS [4] has been developed, in which users can deductively verify safety properties of TLA$^+$ specifications. TLA$^+$ contains a declarative language for writing hierarchical proofs, and TLAPS is built around a *proof manager*, which interprets this proof language, expands the necessary module and operator definitions, generates corresponding proof obligations (POs), and passes them to backend verifiers, as illustrated in Figure 1. While TLAPS is an interactive proof environment that relies on users guiding the proof effort, it integrates automatic backends to discharge proof obligations that users consider trivial.

The two main backends of the current version of TLAPS are Zenon [3], a tableau prover for first-order logic and set theory, and Isabelle/TLA$^+$, a faithful encoding of TLA$^+$ in the Isabelle [11] proof assistant, which provides automated proof methods based on first-order reasoning and rewriting. Zenon is not part of the trusted code base of TLAPS, but outputs proof scripts in Isar syntax for the theorems that it proves. These proofs are passed to Isabelle for verification. In this way, Isabelle/TLA$^+$ is used both as a standalone backend prover and as the certification engine of proof scripts produced by other backends.

The currently available backends also include a generic translation to the input language of SMT solvers, focusing on quantifier-free formulas of linear arithmetic (not shown in Fig. 1). This SMT backend has occasionally been useful because the other verifiers perform quite poorly on obligations involving arithmetic reasoning. However, it covers a rather limited fragment of TLA$^+$, which heavily relies on modeling data using sets and functions. Assertions mixing arithmetic, sets and functions arise frequently in TLA$^+$ proofs. In the work reported here we present a new SMT-based backend for (non-temporal) TLA$^+$ formulas that encompasses set-theoretic expressions, functions, arithmetic, records, and tuples. By evaluating the performance of the backend over several existing TLA$^+$ proofs we show that it achieves good coverage for "trivial" proof obligations.

*Supported by the Microsoft Research-INRIA Joint Centre, Saclay, France.

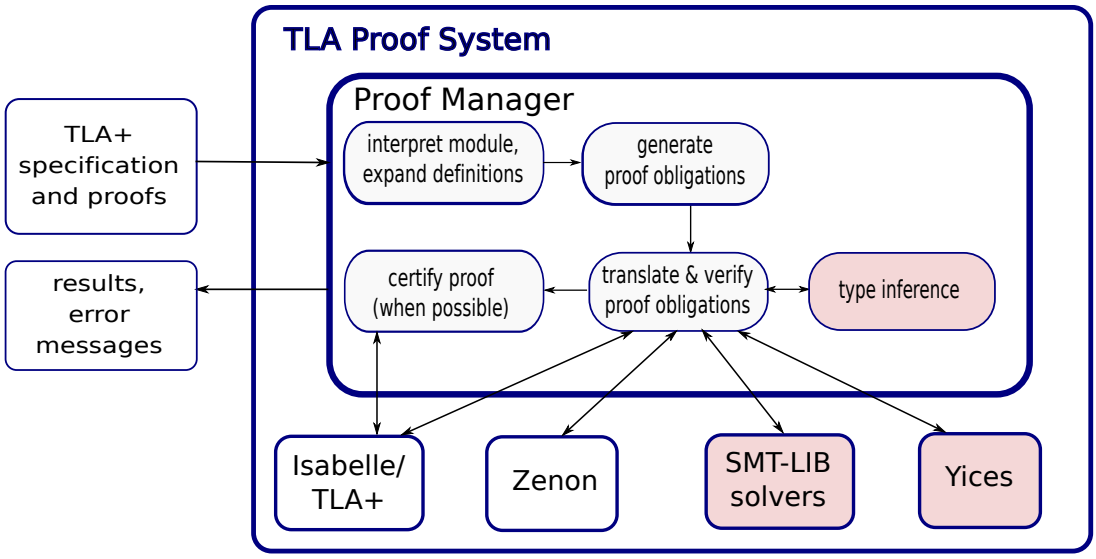

Figure 1: General architecture of TLAPS.

The new modules comprising our backend appear shaded in Figure 1. We consider two target languages for our translation: SMT-LIB [2], the *de facto* standard input format for SMT solvers, and the native input language of the SMT solver Yices [6]. Using SMT-LIB as the target of our translation, TLAPS can be independent of any particular solver. On the other hand, the Yices language provides useful concepts such as sub-typing or a direct representation of tuples and records. The considered TLA$^+$ formulas are translated to quantified first-order formulas over the theory of linear integer and real arithmetic, extended with free sort and function symbols. In particular, we make heavy use of uninterpreted functions, and we do not restrict ourselves to quantifier-free formulas. All SMT solvers mentioned in this paper produce proofs, although each one has its specific output format.

TLA$^+$ is an untyped language, which makes it very expressive and flexible, but also makes automated reasoning quite challenging [9]. Since TLA$^+$ variables can assume *any* value, it is customary (and recommended) to start any verification project by proving a so-called *type invariant* that constrains the values that variables of the specification may assume. Most higher-level correctness proofs rely on the type invariant. It should be noted that TLA$^+$ type invariants frequently express more sophisticated properties than what could be ensured by a decidable type system. In contrast, our target languages are multi-sorted first-order languages, which are supported by dedicated decision procedures in SMT solvers. The first challenge is therefore to assign an SMT sort to each expression that appears in the proof obligation. We make use of this type assignment during the translation of expressions, which may depend on the types of the subexpressions involved. For example, equality between integer expressions will be handled differently from equality between sets or functions.

During a first phase, our translation attempts to infer types for all subexpressions of a proof obligation. This phase may fail because not every set-theoretic expression is typable according to our typing discipline, and in this case the backend aborts. Otherwise, the proof obligation is translated to an SMT formula. Observe that type inference is relevant for the soundness of the SMT backend: a proof obligation that is unprovable according to the semantics of untyped TLA$^+$ must not become provable due to incorrect type annotation. As a trivial example, consider the formula $x + 0 = x$, which should be provable only if x is known to be in an arithmetic domain.

Type inference essentially relies on assumptions that are present in the proof obligation and that constrain the values of symbols (variables or operators).

A type system for TLA$^+$ together with a concise description of the type inference algorithm is presented in the next section. Section 3 describes the translation. Finally, results for some case studies and conclusions will be given in Sections 4 and 5.

2 Type inference for TLA$^+$

We define a type system for TLA$^+$ expressions that underlies our SMT translation. We consider types τ according to the following grammar.

$$\tau ::= i \mid o \mid \mathit{Str} \mid \mathit{Nat} \mid \mathit{Int} \mid \mathit{Real} \mid \mathrm{P}\,\tau \mid \tau \to \tau \mid \mathit{Rec}\,\{\mathit{field}_i \mapsto \tau_i\} \mid \mathit{Tup}\,[\tau_i]$$

The atomic types are i (terms of unspecified type), o (propositions), strings, and natural, integer and real numbers. Complex types are sets (of base type τ), functions, records (defined by a mapping from field names to types) and tuples (as a list of types).

For this type system, we define an inference algorithm that is based on a recursive, bottom-up operator $[\![e, \varepsilon]\!]_I$ whose arguments are a TLA$^+$ expression e and an expected (minimum) type ε that e should at least have, according to a predefined partial order relation on types that includes relations such as $\mathit{Nat} < \mathit{Int}$ or $i < \tau$, for any type $\tau \neq i$. The computation either returns the inferred type or aborts. The operator recurses over the structure of TLA$^+$ expressions, gathering information in a typing environment $\mathit{type} : \mathit{Id} \mapsto \tau$, that maps each TLA$^+$ symbol to its type.

The operator $[\![\cdot]\!]_I$ is applied iteratively on the list of hypotheses $H_1, \ldots, H_n$, until a fixpoint is reached: note that later hypotheses may provide additional information for symbols that occur in earlier ones.

Initially, we consider that every symbol has the unspecified type i, which can be thought of as a bottom type. Recursive calls to the operator $[\![\cdot]\!]_I$ may update the type of symbols as recorded in *type* by new types that are greater than the previous ones. A type assignment is definitive only when types for all expressions in the proof obligation have been successfully inferred. For example, consider the hypotheses $S = \{\}$ and $S \subseteq \mathit{Int}$. After evaluating the first one, S will have type $\mathrm{P}\,i$, but it will be updated to $\mathrm{P}\,\mathit{Int}$ when the second hypothesis is processed.

When type inference succeeds, the environment variable *type* will contain the resulting final type assignments. As we discuss below, there are two cases where the inference algorithm can fail: (1) when a symbol an expression depends on does not have an assigned type, and (2) when a constraint stating that two or more expressions need to be of the same type cannot be solved.

The operator $[\![\cdot]\!]_I$ assigns types to complex expressions based on the types of their constituents. Although expressions such as $\{a\} \cup 0$ or $3 + \text{TRUE}$ appear silly, they are allowed in TLA$^+$, yet their meaning is unknown. Certain TLA$^+$ operators are defined in such a way that the result type is fixed, aiding type inference. For example, logical operators always return Boolean values, whatever are the types of their operands. Similarly, an expression $S \cup T$ is of type $\mathrm{P}\,i$; if S and T are known to be sets of elements of the same type τ, then we obtain the more precise type $\mathrm{P}\,\tau$. Arithmetic operators guarantee the result type only if their arguments are in the arithmetic domain.

TLA$^+$ expressions such as $e_1 = e_2$, $e_1 \subseteq e_2$, IF p THEN e_1 ELSE e_2 are critical in the sense that their constituents e_1 and e_2 must have the same type in order to express them in the sorted format of the SMT solvers. Similarly, the expression $e_1 \in e_2$ requires that e_2 be of type $\mathrm{P}\,\tau$, and e_1 of type τ, for some τ. For similar reasons, we do not allow functions (or operators) to return values of different types. Type inference for this kind of expressions makes use of the function

$\mathcal{S}([e_1, \ldots, e_n], \varepsilon)$, that given a list of expressions $e_1, \ldots, e_n$ and an expected type ε, returns their unique type and fails when some pair $[\![e_1, \varepsilon]\!]_I, \ldots, [\![e_n, \varepsilon]\!]_I$ cannot be equated.

As a concrete example, consider the proof obligation $(\neg\neg P) = P$, where P is a defined operator whose definition is hidden (i.e., its definition cannot be used in the proof obligation). We know that $\neg\neg P$ is Boolean, and the typing rule for equality requires that the expressions on both sides must be of equal type. Without any information on the type of P, the obligation should be rejected by type checking. In particular, we are not allowed to infer that P is of type Boolean. Indeed, P might be defined as the integer 42, and we do not know if $(\neg\neg 42) = 42$ is true in TLA$^+$. We therefore derive the type assignment only from available facts, i.e., hypotheses of the proof obligation, of the forms $id \otimes e$ and $\forall x \in S : id(x) \otimes e$, for $\otimes \in \{=, \in, \subseteq\}$. In these expressions, id is a constant, variable or operator, and e is an expression whose type can already be inferred[1].

3 From TLA$^+$ to SMT

Once we have determined a type assignment for a TLA$^+$ proof obligation, it can be translated to the input languages of SMT solvers. Our target languages (SMT-LIB and Yices) are similar, and the translation proceeds along the same lines. We define the operator $[\![\cdot]\!]_T$ that translates a TLA$^+$ expression, using the type information gathered previously, to its corresponding SMT formula extended with λ-terms. The typing discipline ensures that all resulting λ-abstractions can be β-reduced before obtaining the definitive translation of the original expression.

A symbol's translation depends on its inferred type. It is translated to an SMT variable or function name, which has to be declared in the output produced by the translation, together with its type. Purely arithmetic and first-order expressions are translated to the corresponding built-in operators of the target languages, except for quantified expressions, where quantified variables are introduced temporarily in the context with their types inferred accordingly. In this way, for the expression $\forall x : e$, the value $[\![e, o]\!]_I$ is evaluated to obtain the type assignment for the variable x, so it can be properly declared in the translation of the quantified expression.

Sets and functions are translated to uninterpreted functions. The encoding of a set S represents its characteristic predicate ("is a member of set S"), allowing for the direct translation of the set membership relation. In this way, $[\![S]\!]_T$ is a λ-abstraction which will be applied to a value to detect if it is an element of the set or not, consequently $[\![e \in S]\!]_T \equiv (\![S]\!]_T [\![e]\!]_T)$. Similarly, function application reduces to $[\![f[e]]\!]_T \equiv (\![f]\!]_T [\![e]\!]_T)$. A function $[x \in S \mapsto exp]$, whose domain is S, is translated to $\lambda y.\ [\![exp(x/y)]\!]_T$, where x is replaced by y in the expression exp (the domain S is represented separately, as we will see below).

For example, a function f that returns a set of type P τ it will be translated as a λ-abstraction where the first parameter is the function's argument and the second is a value of type τ. Then the translation is defined as $[\![f]\!]_T \equiv \lambda x\ y.$`(f x y)`, where `f` denotes the symbol's name in the SMT output. The translation of equality depends on the types of the two subexpressions, which must have the same type by type-inference. For example:

$$
\begin{aligned}
[\![e_1 = e_2]\!]_T \equiv [\![\forall x : x \in \text{DOMAIN } e_1 \Leftrightarrow x \in \text{DOMAIN } e_2 \quad & \text{when } e_1 \text{ is of type } \_ \to \_ \\
\wedge\ (x \in \text{DOMAIN } e_1 \Rightarrow e_1[x] = e_2[x])]\!]_T & \\
[\![e_1 = e_2]\!]_T \equiv [\![\forall f, g : (f \in e_1 \wedge g \in e_2) \Rightarrow f = g]\!]_T \quad & \text{when } e_1 \text{ is of type P } (\_ \to \_)
\end{aligned}
$$

[1] For variables, these kind of expressions are usually given by the type invariant. Our backend requires similar type-correctness lemmas for hidden operators.

More generally, n-ary functions or operators returning a (simple) set are represented as predicates of arity $n+1$ with their last argument being the set members. We only consider simple sets, i.e. sets of individuals, in order to remain in the realm of first-order logic. Any hypotheses of a proof obligation that fall outside this class are discarded. For example, hypotheses of the form $x \in S$ where S is of type $\mathrm{PP}\,\tau$ are useful during type inference in order to determine the type of x but are then dropped during the SMT translation.

Because SMT-LIB has no notion of function domain, we associate with each function f a set DOMAIN f (as a characteristic predicate). For every function application that occurs in the proof obligation, we check that the argument values are in the domain: otherwise the value of the function application would be unspecified. To this end, we define an auxiliary operator $[\![\cdot]\!]_F$ that computes corresponding proof obligations. In particular, for function applications we let $[\![f[e]]\!]_F \equiv [\![f]\!]_F \wedge [\![e]\!]_F \wedge e \in \text{DOMAIN } f$ and $[\![Q\,x \in S : e]\!]_F \equiv \forall x \in [\![S]\!]_T : [\![e]\!]_F$ for $Q \in \{\forall, \exists\}$. The remaining clauses in the definition of $[\![\cdot]\!]_F$ collect all function applications that occur in an expression. In particular, $[\![x]\!]_F \equiv \text{TRUE}$ for an atomic expression x. An expression e depending on subexpressions $e_1, \ldots, e_n$, is evaluated as $[\![e(e_1, \ldots, e_n)]\!]_F \equiv [\![e_1]\!]_F \wedge \ldots \wedge [\![e_n]\!]_F$.

The main difference between the SMT-LIB and Yices translations is the encoding of tuples and records. In Yices, they are natively supported, whereas SMT-LIB currently does not have a pre-defined theory. To encode (fixed-size) tuples and records in the first-order logic of SMT-LIB, we treat each of their components separately. A symbol t of type $\mathit{Tup}[\tau_i]$ is translated as $\lambda i.\ t\_i$, introducing the new symbols t_i with types τ_i to the context, corresponding to the i-th component of the tuple t. For example, $[\![t = \langle e_1, e_2 \rangle]\!]_T \equiv [\![t\_1 = e_1 \wedge t\_2 = e_2]\!]_T$. The translation of records is analogous, with field names taking the place of tuple indexes. Currently, all constituents of tuples and records must be of basic types.

4 Experimental results

We have used our new backend with good success on several examples taken from the TLAPS distribution. For example, the non-temporal part of the invariant proof for the well-known N-process Bakery algorithm [7], which mainly uses set theory, functions and arithmetic over the natural numbers, could be reduced from around 320 lines of interactive proof to a completely automatic proof. The resulting obligation generates an SMT-LIB file containing 105 quantifiers (many of them nested), which has been proved by the CVC3 [1] SMT solver in around 10 seconds and by Z3 [5] in less than a second on a standard laptop (whereas the original, interactive proof takes around 24 seconds to process). On the other hand, Yices cannot handle the entire proof obligation at once, and it was necessary to split the theorem into separate cases per subaction; it then takes about 8 seconds to prove the resulting obligations.

More interestingly, the Yices backend (with better support for records) could handle significant parts of the type and safety invariant proofs of the Memoir system [10], a generic framework for executing modules of code in a protected environment. The proofs were almost fully automated, except for three sub-proof that required manual Skolemization of second-order quantifiers. In terms of lines of proof, they were reduced to around 10% of the original size. In particular, the original 2400 lines of proof for the complete type invariant theorems were reduced to 208 lines.

5 Conclusions

We defined a translation of certain TLA$^+$ proof obligations into SMT-LIB 2 and into the native language of Yices. The translation relies on a typing discipline for TLA$^+$, which is untyped, and a corresponding type inference algorithm. This discipline restricts the class of TLA$^+$ expressions that can be translated. Nevertheless, a significant fragment of the source language can be handled. In particular, we support first-order logic, elementary set theory, functions, integer and real arithmetic, tuples and records. Sets and functions are represented as lambda-abstractions, which works quite efficiently but excludes handling second-order expressions involving sets of sets or quantification over complex types. Universal set quantifiers that occur at the outermost level can easily be removed by the user of TLAPS, by introducing Skolem constants. An automatic pre-processing of such terms would further improve the backend. The current SMT-LIB backend provides only limited support for tuples and records.

In future work, we intend to study the question of interpreting proofs provided by SMT solvers for reconstructing them (as well as the type assignment) in the trusted object logic of Isabelle/TLA$^+$. We also envisage extending our translation to the native input languages of other SMT solvers such as Z3 [5]. These are similar to the ones that we considered here, and therefore their translation will be straightforward once the types are assigned.

References

[1] C. Barrett and C. Tinelli. CVC3. In W. Damm and H. Hermanns, editors, 19^{th} *International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298-302. Springer-Verlag, July 2007. Berlin, Germany.

[2] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, eds.: Satisfiability Modulo Theories, 2010.

[3] R. Bonichon, D. Delahaye, and D. Doligez. Zenon : An Extensible Automated Theorem Prover Producing Checkable Proofs. In N. Dershowitz, A. Voronkov, eds.: *14th Intl. Conf. Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2007)*, vol. 4790 of Lecture Notes in Computer Science, pages 151-165, Yerevan, Armenia, 2007. Springer.

[4] K. Chaudhuri, D. Doligez, L. Lamport, and S. Merz. Verifying Safety Properties with the TLA$^+$ Proof System. In J. Giesl and R. Hähnle (eds.): *5th Intl. Joint Conf. Automated Reasoning (IJCAR 2010)*, volume 6173 of *Lecture Notes in Computer Science*, pp. 142-148. Edinburgh, UK, 2010.

[5] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In C. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin / Heidelberg, 2008.

[6] B. Dutertre and L. de Moura. The Yices SMT solver. Tool paper at `http://yices.csl.sri.com/tool-paper.pdf`, 2006

[7] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM*, 17:453-455, August 1974.

[8] L. Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley, 2002.

[9] L. Lamport and L. C. Paulson. Should your specification language be typed? *ACM Trans. Program. Lang. Syst.*, 21:502-526, May 1999.

[10] B. Parno, J. R. Lorch, J. R. Douceur, J. Mickens, and J. M. McCune. Memoir: Practical State Continuity for Protected Modules. *IEEE Symp. Security and Privacy*, IEEE, May 2011. Formal Specifications and Correctness Proofs: Tech. Report, Microsoft Research, Feb. 2011.

[11] M. Wenzel, L. C. Paulson, and T. Nipkow. The Isabelle Framework. In *21st Intl. Conf. Theorem Proving in Higher Order Logics*, TPHOLs '08, pages 33-38, Berlin, Heidelberg, 2008. Springer-Verlag.