Better Automation for TLA+ Proofs

Antoine Defourné

Better Automation for TLA$^+$ Proofs

Antoine Defourné

INRIA

Abstract

TLA$^+$ is a specification language based on traditional untyped set theory. It is equipped with a set of tools, including the TLA$^+$ proof system TLAPS, which uses trusted back-end solvers to handle individual proof steps—referred to as "proof obligations". As most solvers rely on and benefit from *typed* formalisms, types are first reconstructed for the obligations; however, the current encoding into the SMT-LIB format does not exploit all of this type information. In this paper, we present motivations for a more pervasive usage of types at an intermediate representation of TLA$^+$ proof obligations, and describe work in progress on several improvements of TLAPS: a type-driven SMT encoding, a tactic for instantiation hints, and type annotations for the language. We conclude with some perspectives for future work.

1 Introduction

TLA$^+$ is a formal specification language designed by Leslie Lamport [**?**]. It is used to write system specifications, in particular concurrent systems. It is based on the *Temporal Logic of Actions* (TLA)—an extension of first-order logic (FOL) with a notion of state and temporal operators—combined with the operators of *untyped set theory*. This choice of traditional set theory in first-order logic makes TLA$^+$ very similar to the language of usual mathematics, but the absence of types is uncommon among practical formalisms. This design principle is advocated by Lamport and Paulson [**?**] for the flexibility, expressiveness, and ease of use it offers.

Figure 1 shows an example of a specification (on the left). This code specifies sorting algorithms at an abstract level; it leaves out the details of potential implementations, but it could be *refined* by a more concrete specification [**?**]. Several constants are first declared, plus one variable arr, which represents the array to sort. It is initially equal to arrInit, as stated in the predicate Init. The predicate Next contains the primed variable arr' representing the array *in the next state*; that makes Next an *action*, a predicate expressing a relation between two consecutive states. Finally, the predicate Spec assembles the initial predicate and the action into one *temporal formula*. Here the modality $\Box$ prefixes a formula that must be valid across all states. `[][Next]_vars` is an abbreviation for `[](Next \/ vars' = vars)`. Thus Spec states that "Init is initially true, and for all consecutive states either Next is true, or the variables do not change."

TLA$^+$ comes with a set of tools, including the finite model checker TLC [**?**]. This tool can be used to generate behaviors (sequences of states) that satisfy a given specification, usually in the same form than Spec. For example, using TLC we can check that our sort algorithm specification ensures some array gets sorted eventually. This is expressed by the formula Sorted(arr). For a given initial array, we can ask TLC to check that the negation of that property is an invariant; if the specification is correct, TLC will find a sequence of states that lead to a state where the invariant is violated, *ie.* where the array is sorted.

But this approach is limited by the fact that it is only *finite* model-checking. The parameters of the specification must be provided. At best, we can use TLC to analyse the specification for multiple inputs in the finite domain $[1 \cdots N \mapsto 1 \cdots \mathrm{Max}]$, but the domain for arbitrary N

```
(* N: size of the array;
 * Max: maximal value;
 * arrInit: the initial array
 * arr: the array to sort *)
CONSTANTS N, Max, arrInit
VARIABLE arr

vars == <<arr>>
Dom == 1 .. N
USE DEF Dom, vars

ASSUME NIsNat == N ∈Nat
ASSUME MaxIsNat == Max ∈Nat
ASSUME ArrInitIsArray ==
        arrInit  ∈ [ Dom -> 1 .. Max ]

Init == arr = arrInit

Swap(f, i, j) == [ f EXCEPT ![i] = f[j],
                            ![j] = f[i] ]

Next == ∃ i, j ∈ Dom :
           /\ i < j
           /\ arr[i] > arr[j]
           /\ arr' = Swap(arr, i, j)

Spec == Init /\ [][Next]_vars

Sorted(f) == ∀ x, y ∈ DOMAIN f :
               x < y => f[x] <= f[y]
```

```
AXIOM PermId ==
    ∀ f ∈ [ Dom -> 1 .. Max ] : IsPermOf(f, f)
AXIOM PermComp ==
    ∀ f, g ∈ [ Dom -> 1 .. Max ] :
        ∀ i, j ∈ Dom :
            IsPermOf(f, g) => IsPermOf(Swap(f, i, j), g)

THEOREM Spec => []IsPermOf(arr, arrInit)
<1>1 Init => IsPermOf(arr, arrInit)
    <2> HAVE Init
    <2> SUFFICES IsPermOf(arrInit, arrInit)
        BY DEF Init
    <2> QED
        BY PermId, ArrInitIsArray
<1>2 IsPermOf(arr, arrInit) /\ Next
        => IsPermOf(arr', arrInit)
    <2> SUFFICES ASSUME IsPermOf(arr, arrInit), Next
                 PROVE IsPermOf(arr', arrInit)
        OBVIOUS
    <2> SUFFICES ASSUME NEW i ∈Dom,
                        NEW j ∈Dom
                 PROVE IsPermOf(Swap(arr, i, j), arrInit)
        BY DEF Next
    <2> QED
        BY PermComp, ArrInitIsArray
<1>3 IsPermOf(arr, arrInit) /\ arr' = arr
        => IsPermOf(arr', arrInit)
    OBVIOUS
<1> QED
    BY PTL, <1>1, <1>2, <1>3, TypeInv DEF Spec
```

Figure 1: Example of TLA$^+$ Specification and Proof

and Max is infinite. Verifying properties independent of the parameters can only be done by writing and checking a *formal proof*.

The dedicated proof system of TLA$^+$ is TLAPS, developped by the joint center INRIA-Microsoft Research. The main component of TLAPS is its proof manager, which generates several *proof obligations* from a proof script. Each obligation corresponds roughly to an individual proof step, and must be checked by one of TLAPS' trusted back-end solvers.

On the right of figure 1, a proof script is shown. Here a different property of the specification is proved: the array is always a permutation of the initial array[1]. Proofs in TLAPS are hierarchical in the sense that each proof is either one line of relevant facts and definitions, or a sequence of steps, each step justified by its own proof. We have left hidden some parts of the whole script, like the definition of IsPermOf, and some unimportant facts, to lighten the presentation. The structure of this proof is simple and matches a very common pattern: first it is shown that the property holds in the initial state; then that it is preserved by the action (two values are swapped); then that it is also preserved when the array is unchanged; finally, all proof steps are assembled to prove the main goal. The particular method PTL is invoked here to reason about temporal formulas. The difficult steps of the proofs are handled separately in lemmas PermId and PermComp, but we do not present the proofs for those.

[1]Such properties are called "safety properties". They state that some invariant is maintained by the specification. Properties that state that an interesting state is eventually reached (like "the array is eventually sorted") are called "liveness properties".

2 Overview of the SMT Encoding

In previous work [**?**], Vanzetto defined an encoding of proof obligations into SMT-LIB, a standard input format for SMT solvers [**?**]. This permitted to interface TLAPS with CVC4, Zenon and VeriT as back-end solvers.

The language of SMT-LIB is based on multi-sorted first-order logic with equality; the encoding of TLA$^+$ can then be seen as a translation of set theory into that logic. The translation of Vanzetto is composed of two steps: in the first step (preprocessing), the obligation is transformed into an equivalent one which is free of so-called non-basic expressions—that is all expressions that do not have a counterpart in the target logic, mainly set theoretic expressions—the second step is a straightforwad encoding of the basic obligation into SMT-LIB. Most of the work being performed in the preprocessing step, we will present it briefly. Three transformations are applied to the obligation until it is in basic form:

Rewriting applies simple rules that eliminate non-basic expressions directly;

Abstraction replaces non-basic expressions by new constants, which are axiomatised;

Elimination optimizes the translation by simplifying equalities involving variables.

Let $P(x)$ and $Q(x, y, z)$ be two predicates, and a an expression. Consider the following pseudo-example of a preprocessing:

1. $\forall x \forall y.\, P(\underline{\{z \in a : Q(x, y, z)\}})$
2. $(\forall x \forall y.\, P(\mathbf{k}(a, x, y))) \wedge \forall a \forall x \forall y.\, \underline{\mathbf{k}(a, x, y) = \{z \in a : Q(x, y, z)\}}$ (abstraction, $\mathbf{k}$ fresh)
3. $(\forall x \forall y.\, P(\mathbf{k}(a, x, y))) \wedge \forall a \forall x \forall y \forall z.\, z \in \mathbf{k}(a, x, y) \Leftrightarrow z \in a \wedge Q(x, y, z)$ (rewriting)

First the expression $\{z \in a : Q(x, y, z)\}$ is replaced by a fresh constructor $\mathbf{k}$, which is defined at the top-level—for this $\mathbf{k}$ must be parameterized by the base set a and the free variables of the original expression. Then, as the equality between sets matches one of the rewriting rules, it is applied, eliminating the non-basic construct. There is no elimination step in this example, but most eliminated equalities are equalities present in the original obligation.

The procedure above describes a faithful encoding of the untyped expressions of TLA$^+$ into untyped first-order logic, or rather into a unique sort U of multi-sorted FOL. It is limited by the fact that the usual sorts of SMT, like *int*, are left on the side. Consider the goal:

$$\forall x.\, x \in Int \Rightarrow x + 0 = 0$$

The naive encoding would declare a function $+_U$ with signature $U \times U \to U$, and define its behavior with the axioms of arithmetic. But the solvers would not be able to make any use of their reasoning capacities about arithmetic. We can partially remedy this by linking the sorts *int* and U with an injective "cast" operator:

$$\begin{aligned}
&\forall x^U.\, x \in Int \Leftrightarrow \exists n^{int}.\, x = {}^{int}_{U}\!\downarrow n \\
&\forall m^{int} \forall n^{int}.\, \left({}^{int}_{U}\!\downarrow m\right) +_U \left({}^{int}_{U}\!\downarrow n\right) = {}^{int}_{U}\!\downarrow (m + n) \\
&\forall x^U.\, x \in Int \Rightarrow x +_U \left({}^{int}_{U}\!\downarrow 0\right) = x \qquad \text{(Goal)}
\end{aligned}$$

This solution allows solvers to reason about arithmetic, but it obfuscates the resulting problem with casts and additional quantifiers. This was the main motivation in [**?**] for the introduction of *reconstructed types*. Before an obligation is encoded, an algorithm attempts to

infer types for the bound variables. If this succeeds, the process results in an obligation with type annotations, such as:

$$\forall x^{int}.\, x \in Int \Rightarrow x + 0 = 0$$

This obligation can then be encoded more efficiently. In particular, $x \in Int$ can be simplified, and there is no need to introduce a function $+_U$.

The current encoding into the SMT-LIB format implements a rich type system with dependent and refinement types. This allows very fine optimizations of the translation, for example the application $f[x]$ can be translated directly if types ensure the fact $x \in \text{DOMAIN } f$ holds. All expressions of TLA$^+$ are not typable in this way, so the untyped encoding is invoked if the typed encoding fails.

3 Better Encoding of Untyped Set Theory

In this section we present a series of potential improvements of the current encoding of TLA$^+$ into SMT-LIB.

All of these ideas revolve around types, and are somewhat motivated by the assumption that SMT solvers perform best with types in general. Although there are reconstructed types, the encoding we described above does not harness the full potential of this extra type information. It is used for optimizations involving objets of an atomic type, like integers, or to check conditions like membership in some function's domain, but by default the encoding remains untyped.

Type-driven Encoding

Our first proposal is based on the observation that most expressions are still mapped into the generic sort U by the current encoding. In particular, the expression $e_1 \in e_2$ will always lead to some encoded term $\mathsf{in}(e'_1, e'_2)$, where in has signature $\langle U, U\rangle \to o$. In the same vein, functional application $e_1\,[e_2]$ gives $\mathsf{app}(e'_1, e'_2)$ with app of signature $\langle U, U\rangle \to U$. Instead of ignoring types of e_1 and e_2, we propose to extend the encoding in order to specialize the operators in and app according to their operands' types. Let us turn immediately to a concrete example in order to clarify.

$$\begin{aligned}&\text{Surjective}(f, A, B) \triangleq\\ &\quad \forall y \in B.\, \exists x \in A.\, y = f\,[x]\end{aligned}$$

$$\vdash \text{Surjective}([n \in Nat \mapsto n], Nat, Nat)$$

During the type reconstruction phase, bound variables receive type annotations, and defined operators are decorated as well. The problem becomes:

$$\begin{aligned}&\text{Surjective}\ \langle \alpha, \beta\rangle\ (f^{\alpha\to\beta}, A^{\text{Set}(\alpha)}, B^{\text{Set}(\beta)}) \triangleq\\ &\quad \forall y^{\beta} \in B.\, \exists x^{\alpha} \in A.\, y = f\,[x]\end{aligned}$$

$$\vdash \text{Surjective}(\left[n^{int} \in Nat \mapsto n\right], Nat, Nat)$$

Nat is always of type $\text{Set}(int)$, which implies that the bound variable in $[n \in Nat \mapsto n]$ receives the type int, and that the explicit function is of type $int \to int$. The Surjective

operator is polymorphic: it is parameterized by two types α and β. It expects three arguments, all of a higher-order type: a function of type $\alpha \to \beta$, and two sets of respective types $\text{Set}(\alpha)$ and $\text{Set}(\beta)$.

In order to obtain an equivalent problem that is purely first-order, several things need to be done:

- The polymorphic Surjective operator must be specialized. For example, the particular use of this operator in the goal instanciates α with int and β with int.
- Actual objets of a functional type $\tau_1 \to \tau_2$ must be encoded as an object of a fresh first-order sort $\ulcorner \tau_1 \to \tau_2 \urcorner$. Similarly for set-objets.
- Application of a functional variable of type $\tau_1 \to \tau_2$ must be encoded using a specialized operator $\text{Apply}_{\tau_1,\tau_2}$, of type $(\ulcorner \tau_1 \to \tau_2 \urcorner, \tau_1) \to \tau_2$. Similarly for set membership.
- Complex functional or set-theoretic expressions that are translated into uninterpreted operators must be axiomatised.

The final encoded problem should look like:

$$\textsf{New } F_I^{\ulcorner int \to int \urcorner} \tag{Decl I}$$

$$\textsf{New } S_{Nat}^{\ulcorner \text{Set}(int) \urcorner} \tag{Decl Nat}$$

$$\textsf{New } \text{Apply}_{int,int}^{\langle \ulcorner int \to int \urcorner \times int \rangle \to int} \tag{Decl Apply}$$

$$\textsf{New } \text{Mem}_{int}^{\langle \ulcorner \text{Set}(int) \urcorner \times int \rangle \to o} \tag{Decl Mem}$$

$$\forall n^{int}.\, \text{Mem}_{int}(S_{Nat}, n) \Rightarrow \text{Apply}_{int,int}(F_I, n) = n \tag{Def Apply-I}$$

$$\forall n^{int}.\, \text{Mem}_{int}(S_{Nat}, n) \Leftrightarrow n \geq 0 \tag{Def Mem-Nat}$$

$$\text{Surjective}_\Phi(f^{\ulcorner int \to int \urcorner}, A^{\ulcorner \text{Set}(int) \urcorner}, B^{\ulcorner \text{Set}(int) \urcorner}) \triangleq$$
$$\quad \forall y^{int}.\, \text{Mem}_{int}(B, y) \Rightarrow \exists x^{int}.\, \text{Mem}_{int}(A, x) \wedge y = \text{Apply}_{int,int}(f, x)$$

$$\vdash \text{Surjective}_\Phi(F_I, S_{Nat}, S_{Nat})$$

Surjective_Φ is the specialized operator, which expects an argument of type $\ulcorner int \to int \urcorner$ and two arguments of type $\ulcorner \text{Set}(int) \urcorner$. In this example Surjective is specialized only once, but in general there may be several distinct specializations. The set Nat is encoded into the constant S_{Nat} (Decl Nat), then membership for the type $\ulcorner \text{Set}(int) \urcorner$ with the specialized operator Mem_{int} (Decl Mem); finally the membership relation $n \in Nat$ is encoded using an axiom (Def Mem-Nat). The encoding of $\left[n^{int} \in Nat \mapsto n\right]$ leads to similar declarations and definitions (Decl I), (Decl Apply), (Def Apply-I).

This example merely gives an idea of the encoding. We believe previous works such as the defunctionalization procedure described in [**?**] could be adapted to our needs. In particular, the type $\text{Set}(\tau)$ can be treated just like the type of predicates $\tau \to o$. An operator Mem is then merely a particular case of Apply. It is also possible to handle underspecifications, although only partially: in our example, the value of $\text{Apply}_{int,int}(F_I, n)$ is specified under the condition that $\text{Mem}_{int}(S_{Nat}, n)$, that is $n \geq 0$. Thus an expression such as $[n \in Nat \mapsto n]\,[-1]$ would be encoded faithfully. The expression $[n \in Nat \mapsto n]\,[\emptyset]$, however, is likely to make the type reconstruction phase fail.

Instanciation Hints

It is well-known that instantiation of quantifiers is difficult for SMT solvers. As TLAPS relies solely on its trusted back-ends, it can become a problem if an obligation necessitates difficult instantiations–in fact even simple ones sometimes lead to failure. TLAPS provides the keyword "witness" to invoke in a proof script with an expression. A proof step of the form "witness e" in the context of a goal $\exists x.\, \phi(x)$ to prove under hypotheses Γ will result in the obligation $\Gamma, \phi(e) \vdash \exists x.\, \phi(x)$ being generated. Thus the task of matching $\phi(x)$ with $\phi(e)$ still rests on the back-ends.

We propose to exploit a feature of SMT-LIB [**?**, p. 31] to remedy this problem. In SMT-LIB, quantifiers may be annotated with patterns, which are formulas that may trigger an instantiation when a match is detected. Formally, if we consider a formula

$$\forall x_1 \ldots \forall x_n.\, \phi(a_1, \ldots, a_m, x_1, \ldots, x_n)$$

then a pattern for this formula is a list of formulas $(\psi_i)_{1 \leq i \leq p}$ such that the free variables of each ψ_i are contained in $\{a_1, \ldots, a_m, x_1, \ldots, x_n\}$[2]. A match for such a pattern is a substitution θ of the variables such that $\phi_i\theta$ is true for all i. The purpose of a pattern is to suggest that whenever a match θ is found by the SMT solver, the formula should be instantiated with the terms $\theta(x_1), \ldots, \theta(x_n)$. There can be several matches for one quantified formula.

Say that we need to prove a goal that involves instantiating a quantifier with some known expression e. This can happen if the goal is existential, or some hypothesis is universal, for instance. We can do this by declaring a fresh unary predicate W (for "with", or "witness") in the encoded SMT-LIB problem, add the axiom "$W(e)$", and the pattern consisting of the single formula "$W(x)$" for every quantifier Qx that requires to be instantiated. It would seem that this is more patterns than necessary, especially if one single instantiation is aimed for, but we believe that many unnecessary instantiations will still lead to better performances than no instantiations at all.

In the context of a typed encoding, like the one discussed in the previous section, there would be several predicates W_τ, each of type $\tau \to o$, for as much types as necessary. The pattern $W_\tau(x)$ would only annotate quantifiers of the appropriate type Qx^τ. This in fact illustrates the benefit of types: instead of one large domain U, there are several smaller domains, and thus more changes to find the good instances for a quantifier. However, this poses a problem, because for a given instanciation hint e, the type of e may not be known, since e is given outside of any context. Take for instance the following TLA$^+$ problem:

```
Id(S) == [ x ∈ S |-> x ]

THEOREM ASSUME NEW S
        PROVE ∃f ∈ [ S -> S ] : ∀x ∈ S : f[x] = x
   BY DEF Id WITH Id(S)
```

The use of the keyword "with" here is custom—it illustrates how the proposed feature could be implemented in the syntax. The definition of Id need to be expanded in order to prove the property. The issue arises from the fact that no annotations are generated for the term $[x \in S \mapsto x]$ from analysing the proof obligation, since it does not appear in it. But if types were generated for all defined operators in the whole module, which is the subject of the next section, the problem would disappear.

[2] The SMT-LIB standard also specifies that the ψ_i be quantifier-free

Type Annotations

Type reconstruction is currently performed on proof obligations, which are themselves generated from a list of usable facts and definitions. From a single proof script, many obligations can be generated, and all do not share the same amount of information; facts established at some point in a proof may be invoked later, definitions may be hidden or expanded, etc. The "by" and "def" keywords control what facts and definitions should be considered usable in a proof step. The purpose of this simple mechanism is to ensure that obligations do not get saturated with useless information. One major downside is that it is very easy to forget necessary facts that seem irrelevant in a proof. In our experience, this happens quite often with simple facts of the form $x \in S$. For example, $n + 0 = n$ is only true in TLA$^+$ if the fact $n \in Int$ is visible. Unfortunately in many situations n will be known to belong to some set S, and $S \subset Int$ can only be proven if enough definitions are expanded.

This observation lead us to the conclusion that TLAPS could benefit for type annotations at the *module* level. Instead of reconstructing types for each individual obligation, declared operators could be annotated at the top-level, and this information shared for al obligations. We believe this extension will enable new optimizations; returning to our example, if S was an operator defined as $\{n \in Nat : n > 0\}$ (for example), S could be attached the type Set(int), and the fact $S \subset Int$ could be inferred without expanding the definition of S.

As a next step, we also intend to allow the user to provide its own type annotations when declaring operators. A syntax for this feature could be:

```
IsZero(n : int, f : int -> int) : bool == f[n] = 0
```

IsZero would be given the type $(int \times int \to int) \Rightarrow bool$. The double arrow serves to distinguish regular functions from first-order operators, which can only be fully applied.

However, it remains to decide what should be the precise semantics of these annotations, since our intention is not to restrict the class of accepted TLA$^+$ expressions. What should be done if the definition of the operator does not type-check? What if the operator is used with arguments of the wrong type, or in a wrong context, like for instance in the expression IsZero$(\emptyset, [n \in Nat \mapsto 3.14])$?

4 Perspectives and Conclusion

As was said before, we expect these various improvements to result in better performance of the SMT back-ends, and a better ergonomy for TLAPS.

However, as it was assumed all along that types for the obligations could be reconstructed, it remains to decide what should be done when this procedure fails. The validity of so-called silly expressions is a part of TLA$^+$'s design. More importantly, a back-end for TLA$^+$ would not be complete if it prevented us from proving silly theorems, like for instance:

$$\exists n \in Nat : \exists x : x \in n$$

This is true because, if all natural numbers where equal to $\emptyset$, then all natural numbers would be equal to each other, and we would have $0 = 1$.

There is always the possibility to rely on an untyped encoding of TLA$^+$ when the typed encoding fails. But that might lead to a loss in robustness, since one single "anomaly" in the obligation would significantly reduce the chances to solve it, or at least the performance of the solvers.

In place of an optional type system, a *soft* type system would solve this problem. By "soft" we suggest that such a system would be able to infer types for any source expression, and thus not reject any. We are aware of some work [**?**] in that vein in the context of ML, but not in the context of mathematical logic. It is also uncertain if such an approach would lead to a net gain in performance overall, as the type system would be drastically different.

Aside from that, we intend to investigate the possibility of using the reasoning capacities of higher-order solvers directly, notably to improve the support of inductive reasoning. This fall within the project Matryoshka, which aims at extending superposition and SMT solvers with higher-order reasoning in a way that preserves their performances.

5 Acknowledgments

This project receives funding from the European Research Council under the European Union's Horizon 2020 research and innovation program (grant agreement No. 713999, Matryoshka), and the region of Lorraine.