Encoding TLA$^+$ Proof Obligations Safely for SMT

Rosalie Defourné

Encoding TLA$^+$ Proof Obligations Safely for SMT

Rosalie Defourné

Université de Lorraine, CNRS, Inria, LORIA, Nancy, France
rosalie.defourne@inria.fr

Abstract. The TLA$^+$ Proof System (TLAPS) allows users to verify proofs with the support of automated provers, including SMT solvers. To better ensure the soundness of TLAPS, we revisited the encoding of TLA$^+$ into SMT-LIB, whose implementation had become too complex. Our approach is based on a first-order axiomatization with E-matching patterns. The new encoding is available with TLAPS and achieves performances similar to the previous version, despite its simpler design.

Keywords: Automated Theorem Proving · SMT · TLA$^+$ · TLAPS

1 Introduction

TLA$^+$ is a specification language based on the Temporal Logic of Actions and Zermelo-Fraenkel set theory [7, 8, 16]. It is mostly used in the industry for modelling distributed systems [14], but its expressive language is suited for any kind of mathematics [10]. The TLA$^+$ Proof System (TLAPS) provides a syntax for proofs [4]. When a user is satisfied with her proofs, she can invoke TLAPS; the tool will generate a number of proof obligations which are then sent to backend solvers. At this time the solvers available are Isabelle/TLA$^+$ [15], Zenon [2], the SMT solvers CVC4 [1], veriT [3] and Z3 [5], and finally the LS4 prover for temporal logic.

Obligations must be encoded into the respective logics of the selected backends. In this context, a good encoding should meet two requirements: *soundness* and *efficiency*. An efficient encoding makes valid obligations easy for backends to solve. Otherwise users may be forced to reformulate their proofs, which is tedious and time-consuming. Soundness is even more important, as an unsound encoding will let users believe faulty statements are valid. This is especially important for TLAPS as the tool does not verify the solvers' results, with the exception of Zenon, whose output can be checked by Isabelle.

In this paper, we focus on TLAPS's encoding for SMT solvers [12]. To achieve efficiency, the original version of this SMT encoding attempts to simplify away TLA$^+$ primitives. This process is optionally supported by a type synthesis mechanism that assigns sorts to TLA$^+$ subexpressions. Let us illustrate this with the following example:

$$\begin{aligned} &\textsc{assume}\ \ \textsc{new}\ n \in \mathit{Nat} \\ &\textsc{prove}\ \ (0\,..\,n) \cup \{n+1\} = 1\,..\,(n+1) \end{aligned}$$

The expression above is a TLA$^+$ proof obligation. The keyword ASSUME precedes a list of declarations (introduced by NEW) and hypotheses. Here the hypothesis $n \in Nat$ is directly introduced with the declaration of n. The keyword PROVE precedes the goal. Many primitive constructs of TLA$^+$ are standard mathematical notations. "$i \,..\, j$" denotes the set of integers between i and j.

Given this obligation, the original SMT encoding will try to produce an equivalent formula in multi-sorted first-order logic, like this one:

$$\forall n^{\mathsf{int}}.\, n \geq 0 \Rightarrow \forall i^{\mathsf{int}}.\, (1 \leq i \wedge i \leq n) \vee i = (n+1) \Leftrightarrow 1 \leq i \wedge i \leq (n+1)$$

Several techniques are used to achieve this result. A powerful type synthesis mechanism attempts to assign sorts to bound variables—here the builtin sort int of SMT is assigned to n. The obligation is then preprocessed in an attempt to eliminate the TLA$^+$ primitives with no counterpart in SMT. Since n is an integer, both members of the equality are identified as sets of integers, which is why set extensionality is applied. Further rewritings lead to the result displayed. In more complex situations, preprocessing may involve additional techniques like Skolemization of the abstraction of subexpressions.

The original SMT encoding is powerful—in many cases it is able to reduce obligations to trivial problems. But its implementation is very complex and, as a result, difficult to guarantee sound or maintain. There are also limitations inherent to the techniques employed, such as the fact that type synthesis is undecidable, or that simplification may not terminate in some rare cases.

Motivated by the need for a safer encoding, we sought to redesign the SMT encoding in such a way that its most sophisticated features could be disabled. Our original plan was to reimplement type synthesis and simplification, but we found instead that our encoding could be simply optimized with E-matching patterns, also known as "triggers" [6, 11, 13]. This feature of SMT-LIB offers to the user some control over the instantiation of axioms. We claim this is ideal for our purposes, as triggers do not compromise soundness, and TLA$^+$ is naturally formalized using axioms. Our encoding also features axioms for linking TLA$^+$'s integer arithmetic with SMT's, and implements heuristics to find relevant instances of the axiom of set extensionality.

Starting from a formalization of TLA$^+$'s constant fragment (Section 2), we will detail the two essential steps of the encoding: a transformation for recovering formulas (Section 3.2) and then the insertion of axioms (Section 3.3). Our encoding has been implemented in TLAPS, allowing us to compare its performances with the original version (Section 4). Given the simpler design of our encoding, we expected it to perform worse, but we found that performances were similar for the two versions. This suggests that preprocessing TLA$^+$ is not as necessary as we believed to make the SMT encoding efficient: SMT solvers are able to handle the same work if they are provided suitable triggers.

2 Formalizing TLA$^+$'s Constant Fragment

Key Principles A proof of correctness is not possible without a formal definition of TLA$^+$'s semantics. The definition we present is compatible with TLA$^+$'s

reference book [8] and accounts for the addition of lambda-expressions with the second version of the language.[1] We will focus on the *constant fragment* of TLA$^+$, which ignores the temporal aspects of the logic. TLAPS reduces obligations to this fragment during preprocessing. This does not apply to obligations with temporal modalities but, in the current state of TLAPS, we expect these obligations to be isolated from the rest and handled by the prover LS4.

The constant fragment, as a logic, is very close to unsorted first-order logic. It extends the syntax with second-order applications and removes the term-formula distinction. In our formalism, the primitive operators of TLA$^+$ are excluded from the core logic; they are instead declared as part of a standard theory and specified by axioms. This is a convenient way to formalize the underspecified semantics of TLA$^+$. To take one example, the expression

$$\{\emptyset\} \in \mathit{Int} \Rightarrow \{\emptyset\} + 0 = \{\emptyset\}$$

is valid, regardless of the precise interpretation of $\{\emptyset\}+0$. We view this statement as a mere consequence of the axiom

$$\forall x : x \in \mathit{Int} \Rightarrow x + 0 = x$$

Logic without Formulas We define signatures as mappings of operator symbols to types. Types are defined as usual from sorts and a constructor for functional types: the type $\tau = \tau_1 \times \cdots \times \tau_n \to s$ characterizes an operator that takes n arguments and returns an element of sort s. If $n = 0$ then τ is constant and we write $\tau = s$. We define the order $\mathrm{ord}(\tau)$ as 0 in the constant case, else $\max(\mathrm{ord}(\tau_i))_{1 \leq i \leq n} + 1$. If $\mathrm{ord}(\tau) \leq 1$ then n is called the arity of τ.

Definition 1 (Expressions). *We note ι the sort of individuals. A TLA$^+$ signature is a signature Σ such that, for all k, the type $\Sigma(k)$ has order 2 at most and only includes the sort ι. Given such a Σ, the syntax of TLA$^+$ expressions and arguments is defined by the following syntax:*

$$e ::= x \mid k(f, \ldots, f) \mid e = e \mid \text{FALSE} \mid e \Rightarrow e \mid \forall x : e \qquad \text{(Expressions)}$$

$$f ::= e \mid k \mid \lambda x, \ldots, x : e \qquad \text{(Arguments)}$$

where x is a variable symbol and k an operator symbol in the domain of Σ. We impose $\mathrm{ord}(\Sigma(k)) = 1$ if k occurs as an argument. All applications $k(f_1, \ldots, f_n)$ must be well-formed: the arity of f_i must match the arity of the expected type τ_i.

The grammar above defines a minimal fragment of the syntax. The logical connectives TRUE, $\neq$, $\neg$, $\wedge$, $\vee$, $\Leftrightarrow$, $\exists$ may be defined as notations. Note that lambda-expressions may only appear as arguments to second-order operators. Note also that the notion of predicate symbol is absent, much like the notion of formula.

The definition of interpretations is not standard, but still very close to the traditional one for first-order logic. We introduce it briefly; the full definition

[1] http://lamport.azurewebsites.net/tla/tla2-guide.pdf

can be found in appendix A. A domain is a collection D that contains at least two values $\top^D$ and $\bot^D$. An interpretation $\mathcal{I}$ consists of a domain and a mapping $k \mapsto k^{\mathcal{I}}$. The evaluation of expressions e and arguments f is defined recursively such that $[\![e]\!]^{\mathcal{I}}$ is an element of D and $[\![f]\!]^{\mathcal{I}}$ is a function from D^n to D where f is n-ary. For example, the implication case states:

$$[\![e_1 \Rightarrow e_2]\!]^{\mathcal{I}} \triangleq \begin{cases} \top^D & \text{if } [\![e_1]\!]^{\mathcal{I}} \neq \top^D \text{ or } [\![e_2]\!]^{\mathcal{I}} = \top^D \\ \bot^D & \text{otherwise} \end{cases}$$

The satisfaction relation is defined by $\mathcal{I} \models e$ iff $[\![e]\!]^{\mathcal{I}} = \top^D$. Remark that this definition makes $e \Rightarrow e$ a tautology for all e. The two key ideas of the semantics are: Boolean connectives and equality always return Boolean values; if e occurs where a Boolean is expected, $[\![e]\!]$ is compared with $\top^D$ to obtain a Boolean.

Primitive Operators TLA$^+$ defines primitive constructs for many kinds of data including sets, functions, integers and reals. We view all of these constructs as special cases of the application $k(f_1, \ldots, f_n)$. For instance, the TLA$^+$ expression $x \in y$ will be represented by $mem(x, y)$. The operator mem is declared with the type $\iota \times \iota \to \iota$. Note that the lack of a Boolean sort makes it impossible to declare mem as a predicate.

Constructs that bind a variable may be represented with second-order applications. For instance, the set $\{x \in S : e\}$ is represented by $setst(S, \lambda x : e)$, where $setst : \iota \times (\iota \to \iota) \to \iota$. Again, it is not possible to specify that $setst$ expects a predicate argument. The other second-order constructs of TLA$^+$ are the choose expression CHOOSE $x : e$, the replacement set $\{e : x \in S\}$, and the explicit function $[x \in S \mapsto e]$.

The operators of TLA$^+$ are specified by axioms. For instance, the following schema of comprehension holds for all unary P:

$$\forall a, x : mem(x, setst(a, P)) \Leftrightarrow mem(x, a) \wedge P(x)$$

We do not present the axioms here. They are easy to infer from the reference book, and most of them are standard (notably the axioms of ZF). For an explicit presentation of TLA$^+$'s axioms, we refer the reader to our documentation.[2] Since our encoding inserts axioms directly in the SMT problem, the section about axiomatization will feature examples (Section 3.3).

3 Encoding TLA$^+$ for SMT

3.1 Overview

Let us go back to the example from the introduction. With our formalism, we might want to rewrite the obligation as follows:

ASSUME NEW p, $mem(p, Nat)$
PROVE $cup(range(1, p), enum_1(plus(p, 1))) = range(1, plus(p, 1))$

[2] https://github.com/adef-inr/tlaplus-axioms

Every operator is implicitly assigned a type with the single sort ι. For instance, $enum_1 : \iota \to \iota$ and $plus : \iota \times \iota \to \iota$. This applies to the constant operators as well. Thus we have $1 : \iota$.

The first step of the encoding is to recover formulas. The sort o is introduced, and the usual semantics for Boolean connectives is recovered. Equalities are considered formulas as well. It is sometimes necessary to insert conversions; a new operator $cast_o : o \to \iota$ is introduced in the signature for this. This example happens to be left unchanged by the transformation, except for the fact that mem is reassigned the type $\iota \times \iota \to o$.

A simple example of an expression that must be changed is $\text{TRUE} \in \text{BOOLEAN}$. We consider that $\text{TRUE} : o$ in the target logic. But set membership is defined on ι, so the encoding would insert a cast, resulting in $cast_o(\text{TRUE}) \in \text{BOOLEAN}$. Here is a more complex example: in the expression $n \in Nat \Rightarrow p[n]$, the subexpression $p[n]$ is not clearly Boolean, so it is converted into a formula. The result is the formula $n \in Nat \Rightarrow (p[n] = cast_o(\text{TRUE}))$.

The next step, axiomatization, simply inserts explicit declarations and axioms for the relevant TLA$^+$ primitives. Our method of axiom selection is straightforward. Each operator is assigned a set of axioms, which are all inserted after its declaration. If an axiom features an operator not declared yet, the process is repeated recursively. For instance, our example features the operator for taking the union of two sets, cup, which is specified by the axiom

$$\begin{aligned} &\forall a^\iota, b^\iota, x^\iota : \{mem(x, cup(a, b))\} \\ &\quad mem(x, cup(a, b)) \Leftrightarrow mem(x, a) \vee mem(x, b) \end{aligned}$$

Note the expression between curly braces, which is an example of a trigger. It is a hint for SMT solvers to indicate how to instantiate the axiom. Triggers are essential in the optimization of the encoding.

Our target logic includes SMT's builtin sort int. In order to take advantage of SMT's reasoning techniques for integer arithmetic, we treat TLA$^+$'s integer primitives specially. This involves the addition of an injector $cast_{\mathsf{int}} : \mathsf{int} \to \iota$ into the signature. This will be described in more details later; for now, let us simply mention that the integer constants of TLA$^+$ can be encoded as their counterparts in int using casts. In our example, 1 is rewritten to $cast_{\mathsf{int}}(1)$.

The final result, with types made explicit, may be written:

$$\begin{aligned} \text{ASSUME}\ &\text{NEW}\ cast_o : o \to \iota,\ \text{NEW}\ cast_{\mathsf{int}} : \mathsf{int} \to o, \\ &\text{NEW}\ mem : \iota \times \iota \to o,\ \text{NEW}\ cup : \iota \times \iota \to \iota,\ \text{NEW}\ enum_1 : \iota \to \iota \\ &\ldots \qquad \text{(other declarations + axioms with triggers)} \\ &\text{NEW}\ p : \iota,\ mem(p, Nat) \\ \text{PROVE}\ &cup(range(cast_{\mathsf{int}}(1), p), enum_1(plus(p, cast_{\mathsf{int}}(1)))) \\ &\quad = range(cast_{\mathsf{int}}(1), plus(p, cast_{\mathsf{int}}(1))) \end{aligned}$$

At this point, the obligation can be directly translated to SMT. This short overview does not cover two difficult points, which are the reduction of second-

order applications to first-order ones, and our support for set extensionality. These points will be addressed in the section about axiomatization.

3.2 Recovering Formulas

Intuitively, the usual distinction between terms and formulas can be recovered by inserting appropriate conversions in TLA$^+$ expressions. We define a transformation $\mathcal{B}^o$ from TLA$^+$'s core logic to a logic that features the sort o, interpreted as the domain of truth values, and enjoy the traditional semantics for Boolean connectives and equality. Using a new operator $cast_o$ with type $o \to \iota$, we describe two kinds of conversions:

$$e \longrightarrow cast_o(e) \qquad \text{(Injection)}$$
$$e \longrightarrow e = cast_o(\text{TRUE}) \qquad \text{(Projection)}$$

Expressions that appear to be formulas but occur in a non-Boolean context are injected into ι. Conversely, expressions that do not appear to be formulas but occur in a Boolean context are projected onto o. This is illustrated by the example below (which is a valid expression):

$$\forall x : (x = \text{FALSE}) \Rightarrow \neg x \quad \xrightarrow{\mathcal{B}^o} \quad \forall x^\iota : (x = cast_o(\text{FALSE})) \Rightarrow \neg(x = cast_o(\text{TRUE}))$$

We annotate bound variables with sorts in the target logic. This is mostly to emphasize the fact that output formulas belong to a different logic. All bound variables are annotated with ι.

Formal Definition The target logic features the two sorts ι and o. The syntax is now restricted as usual, for instance FALSE has sort o and $e_1 \Rightarrow e_2$ is well-typed with o only if e_1 and e_2 have type o. The interpretation of Boolean connectives is also the standard one. The sort o is interpreted as the collection whose elements are $\top$ and $\bot$.

Given a TLA$^+$ signature Σ, we define $\Sigma^{\mathcal{B}}$ by adding $cast_o$ with type $o \to \iota$. All other operators are preserved with their types. The mappings defined below take their inputs from the core logic of TLA$^+$ under Σ and return terms, formulas or arguments in the target logic just described, under the signature $\Sigma^{\mathcal{B}}$.

Definition 2. *We define by mutual recursion the mappings $\mathcal{B}^\iota$ and $\mathcal{B}^o$ on expressions and $\mathcal{B}^f$ on arguments:*

$$\mathcal{B}^\iota(x) \triangleq x$$
$$\mathcal{B}^\iota(k(f_1, \ldots, f_n)) \triangleq k(\mathcal{B}^f(f_1), \ldots, \mathcal{B}^f(f_n))$$
$$\mathcal{B}^\iota(e) \triangleq cast_o(\mathcal{B}^o(e))$$

$$\mathcal{B}^o(e_1 = e_2) \triangleq \mathcal{B}^\iota(e_1) = \mathcal{B}^\iota(e_2)$$
$$\mathcal{B}^o(\text{FALSE}) \triangleq \text{FALSE}$$
$$\mathcal{B}^o(e_1 \Rightarrow e_2) \triangleq \mathcal{B}^o(e_1) \Rightarrow \mathcal{B}^o(e_2)$$
$$\mathcal{B}^o(\forall x : e) \triangleq \forall x^\iota : \mathcal{B}^o(e)$$
$$\mathcal{B}^o(e) \triangleq \mathcal{B}^\iota(e) = cast_o(\text{TRUE})$$

$$\mathcal{B}^f(e) \triangleq \mathcal{B}^\iota(e)$$
$$\mathcal{B}^f(k) \triangleq k$$
$$\mathcal{B}^f(\lambda x_1, \ldots, x_n : e) \triangleq \lambda x_1^\iota, \ldots, x_n^\iota : \mathcal{B}^\iota(e)$$

The last equations for $\mathcal{B}^\iota$ and $\mathcal{B}^o$ are respectively called injection *and* projection. *They are applied with lowest priority to ensure termination.*

The definition above is not obviously inductive, but we may reason by induction on the construction of any $\mathcal{B}^\iota(e)$, $\mathcal{B}^o(e)$ or $\mathcal{B}^f(f)$. This is justified by the fact that an injection can never immediately follow a projection, or vice versa. If, for example, $\mathcal{B}^\iota(e)$ is obtained by injecting $\mathcal{B}^o(e)$ into ι, then $\mathcal{B}^o(e)$ can only be constructed by applying $\mathcal{B}^\iota$ or $\mathcal{B}^o$ to subexpressions of e.

It is easy to verify that the three mappings result in well-typed expressions. $\mathcal{B}^\iota$ results in terms of the sort ι. $\mathcal{B}^o$ results in formulas of the sort o. If f has arity n, then $\mathcal{B}^f(f)$ has the n-ary type $\iota \times \cdots \times \iota \to \iota$.

Correctness The main result is the theorem 1 below, which is about how each mapping preserves evaluation. We only provide a sketch of the proof here; the full version can be found in appendix A.

For all Σ-interpretation $\mathcal{I}$, we define a $\Sigma^{\mathcal{B}}$-interpretation $\mathcal{I}^{\mathcal{B}}$ by adding an interpretation for $cast_o$. The function $cast_o^{\mathcal{I}^{\mathcal{B}}}$ maps $\top$ to $\top^D$ and $\bot$ to $\bot^D$. The domain D is preserved and the interpretations of all operators in Σ as well.

Theorem 1. *Let $\mathcal{I}$ be a TLA$^+$ interpretation. The following propositions hold for all expressions e and arguments f:*

i) $[\![\mathcal{B}^\iota(e)]\!]^{\mathcal{I}^{\mathcal{B}}} = [\![e]\!]^{\mathcal{I}}$
ii) $[\![\mathcal{B}^o(e)]\!]^{\mathcal{I}^{\mathcal{B}}} = \top$ *iff* $[\![e]\!]^{\mathcal{I}} = \top^D$
iii) $[\![\mathcal{B}^o(e)]\!]^{\mathcal{I}^{\mathcal{B}}} = \bot$ *implies* $[\![e]\!]^{\mathcal{I}} = \bot^D$ *when $\mathcal{B}^o(e)$ is not a projection*
iv) $[\![\mathcal{B}^f(f)]\!]^{\mathcal{I}^{\mathcal{B}}} = [\![f]\!]^{\mathcal{I}}$

Proof. The proof is by induction on the construction of the result. For cases constructing $\mathcal{B}^\iota(e)$, we prove i). For cases constructing $\mathcal{B}^f(f)$, we prove iv). For cases constructing $\mathcal{B}^o(e)$, we prove ii) and iii), except in the case of projection, where only ii) needs to be proved. When an induction hypothesis on $\mathcal{B}^o(e)$ must be invoked, we may only use ii) in general. However, in the case of injections, we use the fact that the previous rule cannot be a projection, so iii) can be used.

Soundness follows trivially from theorem 1. Completeness also follows if the mapping $\mathcal{I} \mapsto \mathcal{I}^{\mathcal{B}}$ is surjective. This is actually not the case, as we could have a domain D in the target logic with only one element; D would not be a suitable domain for TLA$^+$ since we must have $\top^D \neq \bot^D$. We simply exclude this case with the following axiom, which essentially specifies $cast_o$ as injective:

$$cast_o(\text{TRUE}) \neq cast_o(\text{FALSE}) \tag{B}$$

Theorem 2 (Soundness and Completeness of $\mathcal{B}^o$). *Let e be a TLA$^+$ expression. Then e is satisfiable iff $\mathcal{B}^o(e)$ is satisfiable by a model of* (B).

Proof. If $\mathcal{I} \models e$ then $\mathcal{I}^{\mathcal{B}} \models \mathcal{B}^o(e)$ by theorem 1. Clearly $\mathcal{I}^{\mathcal{B}}$ satisfies (B). Conversely, if $\mathcal{J} \models \mathcal{B}^o(e)$ with $\mathcal{J}$ model of (B), then we define $\top^D \triangleq [\![cast_o(\text{TRUE})]\!]^{\mathcal{J}}$ and $\bot^D \triangleq [\![cast_o(\text{FALSE})]\!]^{\mathcal{J}}$. Let $\mathcal{I}$ be the restriction of $\mathcal{J}$ that ignores $cast_o$. It is clear that $\mathcal{J} = \mathcal{I}^{\mathcal{B}}$, so $\mathcal{I} \models e$ by theorem 1. □

Assigning Predicate Types to TLA$^+$ Primitives The encoding $\mathcal{B}^o$ just described preserves the types of all operators with the sort ι. In reality, some reassignments using the sort o are justified. For example, we give set membership *mem* the new type $\iota \times \iota \to o$, and set comprehension *setst* the type $\iota \times (\iota \to o) \to \iota$. We also consider that the axiom schema of set comprehension should be

$$\forall a^\iota, x^\iota : mem(x, setst(a, P)) \Leftrightarrow mem(x, a) \wedge P(x)$$

for all unary *predicate* P. In contrast, applying $\mathcal{B}^o$ to the original axiom schema would introduce a number of conversions from and to o.

The justifications for these type reassignments stem from the semantics of the relevant primitives. Briefly, *mem* may be assigned a predicate type because TLA$^+$ specifies that set membership always returns a Boolean value. Our encoding actually implements the rules:

$$\mathcal{B}^o(mem(e_1, e_2)) \triangleq mem(\mathcal{B}^\iota(e), \mathcal{B}^\iota(e))$$
$$\mathcal{B}^\iota(mem(e_1, e_2)) \triangleq cast_o(\mathcal{B}^o(mem(e_1, e_2)))$$

where $mem : \iota \times \iota \to o$ in the signature $\Sigma^{\mathcal{B}}$. The encoding may be adapted in similar ways, and for similar reasons, to assign predicate types to the subset relation and all the comparison operators of arithmetic.

For set comprehension, the argument is a little more complex. The following equality is valid in TLA$^+$ for all e_1 and e_2:

$$\{x \in e_1 : e_2\} = \{x \in e_1 : e_2 = \text{TRUE}\}$$

But note that this is due to set extensionality, because the expressions e_2 and $e_2 = \text{TRUE}$ are equivalent. This gives the intuition for why we may project the second argument as a predicate. The rule we implement is:

$$\mathcal{B}^\iota(setst(e_1, \lambda x : e_2)) \triangleq setst(e_1, \lambda x^\iota : \mathcal{B}^o(e_2))$$

where $setst : \iota \times (\iota \to o) \to \iota$ in $\Sigma^{\mathcal{B}}$. The rule and justification for assigning $choose : (\iota \to o) \to \iota$ is analogous, but the principle of extensionality for choice is invoked instead.

3.3 Axiomatization

The principle of this step is to make explicit declarations for relevant TLA$^+$ primitives and insert their axioms in the final obligation. The vast majority of axioms are just reformulations of TLA$^+$'s theory. The only exception is our axioms for integer arithmetic, which introduce the sort int, but it will be clear that their inclusion does not compromise soundness.

Our method of axiom selection is straightforward. A declaration is inserted for every primitive that occurs in the obligation. Each primitive may be assigned a number of axioms (typically 1–3) which are inserted in the problem after the

declaration. The process is recursively repeated if axioms contain primitives that are not declared yet.

Here is an example of an axiom with a trigger:

$$\forall a^\iota, b^\iota, x^\iota : \{mem(x, cap(a, b))\}$$
$$mem(x, cap(a, b)) \Leftrightarrow mem(x, a) \land mem(x, b)$$

A trigger is a list of terms annotating the body of a universally quantified formula. We write them between curly braces. Triggers do not affect the semantics of axioms, but SMT solvers may use them to select instances based on the terms that are *known* at a given moment. For instance, when a formula $mem(t_1, cap(t_2, t_3))$ is found, the match $\{x \mapsto t_1, a \mapsto t_2, b \mapsto t_3\}$ may be used to generate an instance of the axiom above. Triggers may include several terms, in which case all terms must match at the same time. Axioms may include several triggers, in which case any individual trigger can produce an instance.

Some SMT solvers implement heuristics for generating triggers, but we found that we could solve more problems by selecting our own triggers cautiously. In the next part of this section, we illustrate some principles behind our methodology through an example. We lack the space for a full presentation of the theory, which includes 80 axioms in total; the complete list can be found in our documentation.[3] After this discussion, we present our solutions for handling integer arithmetic, second-order operators, and set extensionality.

Selecting Triggers for Set Theory For this part, we will use the TLA$^+$ obligation displayed on the left below. The same problem is displayed on the right using our standard notation; the operators *mem* and *subseteq* are predicates, the constant 1 is SMT's builtin integer constant and $cast_{\mathsf{int}} : \mathsf{int} \to \iota$.

ASSUME NEW S,	ASSUME NEW $S : \iota$,
$(S \cap Int) \subseteq \emptyset$	$subseteq(cap(S, Int), empty)$
PROVE $1 \notin S$	PROVE $\neg mem(cast_{\mathsf{int}}(1), S)$

When the problem is translated to SMT, the goal is negated; the obligation will be solved if the SMT solver answers "unsatisfiable". So we may consider $1 \in S$ to be an assumption. The objective is to derive a contradiction. The intuitive proof is that $1 \in S$ and $1 \in Int$ entail $1 \in (S \cap Int)$, but then $1 \in \emptyset$ by inclusion, and a contradiction is derived.

For this example, we will focus on the axioms for *subseteq* and *cap*. The axiom for *empty* is not particularly insightful. The axioms for *Int* and $cast_{\mathsf{int}}$ are discussed later. We may assume that $mem(cast_{\mathsf{int}}(1), Int)$ is derived immediately by SMT and that the contradiction is found when $mem(cast_{\mathsf{int}}(1), empty)$ is

[3] https://github.com/adef-inr/tlaplus-axioms

derived. Here is a first attempt at an axiomatization:

$$\forall a^\iota, b^\iota : \{subseteq(a,b)\} \qquad \text{(SUBSETEQ)}$$
$$subseteq(a,b) \Leftrightarrow (\forall x^\iota : mem(x,a) \Rightarrow mem(x,b))$$
$$\forall a^\iota, b^\iota, x^\iota : \{mem(x, cap(a,b))\} \qquad \text{(CAP)}$$
$$mem(x, cap(a,b)) \Leftrightarrow mem(x,a) \wedge mem(x,b)$$

This attempt is natural if one thinks of triggers as a way of implementing rewriting rules: for both axioms, the left member of the equivalence is given as sole trigger. In our case, the definition for $(S \cap Int) \subseteq \emptyset$ is generated; this amounts to inserting the fact

$$\forall x^\iota : mem(x, cap(S, Int)) \Rightarrow mem(x, empty)$$

in the problem. The next step is to instantiate this new fact with $cast_{\mathsf{int}}(1)$.

But note that the quantifier $\forall x^\iota$ does not have a trigger. As a result, SMT must find the correct instance by other means. As obligations get larger, it becomes increasingly harder for SMT to find the right instances without indications. The solution is to avoid axioms that introduce universal quantifiers in the problem. The axiom (SUBSETEQ) is easily reformulated by breaking down the equivalence in two implications, resulting in two new axioms. For one of them, the universal quantifier can be moved up and a better trigger can be selected:

$$\forall a^\iota, b^\iota : \{subseteq(a,b)\} \qquad \text{(SUBSETEQINTRO)}$$
$$(\forall x^\iota : mem(x,a) \Rightarrow mem(x,b)) \Rightarrow subseteq(a,b)$$
$$\forall a^\iota, b^\iota, x^\iota : \{subseteq(a,b), mem(x,a)\} \qquad \text{(SUBSETEQELIM)}$$
$$subseteq(a,b) \wedge mem(x,a) \Rightarrow mem(x,b)$$

The quantifier $\forall x^\iota$ in (SUBSETEQINTRO) is viewed as existential, as it occurs in a negative context (on the left of an implication). There is no need to assign it a trigger.

We now need two formulas to trigger (SUBSETEQELIM). The assumption $(S \cap Int) \subseteq \emptyset$ is again relevant; the second formula we need is $1 \in (S \cap Int)$. But we have no way of generating that formula: our axiom (CAP) has one trigger, which expects exactly the formula we want to generate.

The trigger of (CAP) can only generate the definition of a formula $x \in (a \cap b)$ that is already known. If we want to use the axiom to generate the formula $x \in (a \cap b)$ instead, we need another trigger. Let us already rule out the candidate

$$\{mem(x,a), mem(x,b)\}$$

That trigger would indeed use the known facts $1 \in S$ and $1 \in Int$ and generate $1 \in (S \cap Int)$. The problem is that, in general, instantiating axiom (CAP) with that trigger can introduce a term $a \cap b$ into the problem. A recurring challenge when selecting triggers is to prevent situations in which axioms may trigger each others indefinitely; but this would happen here. Given any formula $x \in y$

known at a given moment, it is clear that (Cap) could keep triggering itself by matching the same formula twice, producing the formulas $x \in (y \cap y)$, then $x \in ((y \cap y) \cap (y \cap y))$, and so on.

The correct solution is to add two triggers to the axiom, as follows:

$$\begin{aligned} \forall a^\iota, b^\iota, x^\iota : \ & \{mem(x, cap(a,b))\} \\ & \{mem(x,a), cap(a,b)\} \\ & \{mem(x,b), cap(a,b)\} \\ & mem(x, cap(a,b)) \Leftrightarrow mem(x,a) \wedge mem(x,b) \end{aligned} \tag{Cap'}$$

The second trigger above will match the assumption $1 \in S$ and the known term $S \cap Int$. Equivalently, the third trigger can match the assumption $1 \in Int$ and the same term, for the same result.

We have now arrived at an axiomatization that allows SMT solvers to prove the original obligations using only triggers. To summarize the proof: first the axiom (Cap') is triggered by $mem(cast_{\mathsf{int}}(1), S)$ and $cap(S, Int)$, generating

$$mem(cast_{\mathsf{int}}(1), cap(S, Int)) \Leftrightarrow mem(cast_{\mathsf{int}}(1), S) \wedge mem(cast_{\mathsf{int}}(1), Int)$$

Then the axiom (SubseteqElim) is triggered by $mem(cast_{\mathsf{int}}(1), cap(S, Int))$ and $subseteq(cap(S, Int), empty)$, resulting in

$$\begin{aligned} & subseteq(cap(S, Int), empty) \wedge mem(cast_{\mathsf{int}}(1), cap(S, Int)) \\ & \quad \Rightarrow mem(cast_{\mathsf{int}}(1), empty) \end{aligned}$$

From here the contradiction is obtained using propositional logic.

General Principles for Selecting Triggers We systematically reformulate the axioms that feature nested quantifier, so that all universal quantifiers can be moved at the top. This prevents the introduction of quantifiers without triggers during solving, and usually invites us to select different triggers. In particular, many axioms feature an equivalence where one member contains a quantifier, in which case the equivalence is broken down in two implications, resulting in an introduction and an elimination axiom.

The next important idea is to observe what kinds of terms can be generated for a given axiom and trigger. When looking at the axiom for *cap*, we rejected the trigger than could lead to the generation of more terms $a \cap b$, but we kept the triggers that could only generate set membership statements. This illustrates our following pragmatic assumption about TLA$^+$ and its usage: even though the language is very expressive, and obligations may feature complex set expressions, we assume that all the sets relevant to the proof are already in the obligation. However, proofs may rely on many set membership facts that are only implicit. In our example, $1 \in (S \cap Int)$ was such a fact. The element 1 and the set $S \cap Int$ where explicit in the obligation, but their relationship was not.

We have applied a similar principle for functions, only instead of sets, we assume that obligations never require constructing explicit functions $[x \in S \mapsto e]$

or functional sets $[a \to b]$ other than the ones already explicit. We do generate terms like DOMAIN f and $f[x]$ for the known functions f and elements x in their domains. This is illustrated by the axiom below, which is only one component of the definition of $[a \to b]$, also written $arrow(a, b)$. Both triggers need a fact $f \in [a \to b]$ and both may generate the fact $f[x] \in b$ (where $f[x]$ is written $fcnapp(f, x)$). The first trigger may generate a term $f[x]$, while the second may generate a formula $x \in a$.

$$\begin{aligned}\forall a^\iota, b^\iota, f^\iota, x^\iota : \; & \{mem(f, arrow(a, b)), mem(x, a)\} \\ & \{mem(f, arrow(a, b)), fcnapp(f, x)\} \\ mem(f, arrow(a, b)) \wedge mem(x, a) \Rightarrow\; & mem(fcnapp(f, x), b)\end{aligned}$$

Axioms for Integer Arithmetic The construction we describe here was already present in the previous SMT encoding. Its purpose is to link TLA$^+$'s arithmetic with SMT's builtin arithmetic, in order to reason more efficiently on integers. The intuition is that the predicate $n \in Int$ can be made to correspond with the sort int through a simple construction involving the injector $cast_{\mathsf{int}} : \mathsf{int} \to \iota$. We specify its left-inverse $proj_{\mathsf{int}} : \iota \to \mathsf{int}$. This is a well-known trick to specify a function as injective with a simpler axiom. Finally, we specify $cast_{\mathsf{int}}$ as a homomorphism between the arithmetical symbols of both logics. The example below includes the necessary axioms for handling the TLA$^+$ primitives Int and $+$.

$$cast_{\mathsf{int}} : \mathsf{int} \to \iota$$

$$proj_{\mathsf{int}} : \iota \to \mathsf{int}$$

$$\forall z^{\mathsf{int}} : \{cast_{\mathsf{int}}(z)\} \;\; mem(cast_{\mathsf{int}}(z), Int) \qquad \text{(IntIntro)}$$

$$\forall x^\iota : \{mem(x, Int)\} \;\; mem(x, Int) \Rightarrow x = cast_{\mathsf{int}}(proj_{\mathsf{int}}(x)) \qquad \text{(IntElim)}$$

$$\forall z^{\mathsf{int}} : \{cast_{\mathsf{int}}(z)\} \;\; z = proj_{\mathsf{int}}(cast_{\mathsf{int}}(z)) \qquad \text{(IntCast)}$$

$$\begin{aligned}\forall z_1^{\mathsf{int}}, z_2^{\mathsf{int}} : \; & \{plus(cast_{\mathsf{int}}(z_1), cast_{\mathsf{int}}(z_2))\} \\ & plus(cast_{\mathsf{int}}(z_1), cast_{\mathsf{int}}(z_2)) = cast_{\mathsf{int}}(z_1 + z_2)\end{aligned} \qquad \text{(IntPlus)}$$

The axioms for all other operators are analogous to IntPlus. For constants, the axiom is a trivial equality; we simply rewrite the TLA$^+$ constants 0, 1, 2 directly as $cast_{\mathsf{int}}(0)$, $cast_{\mathsf{int}}(1)$, $cast_{\mathsf{int}}(2)$.

Those axioms are not derived from TLA$^+$'s theory, but extend it conservatively. The soundness of the construction relies on the fact that TLA$^+$'s arithmetic and SMT's are assumed to be equivalent. More precisely: from every proposition with ι and int that is valid according to SMT, one obtains a valid TLA$^+$ formula by relativizing all quantifier on int with the predicate $n \in Int$.

Elimination of Second-order Applications Second-order applications are reduced to first-order ones during this step. The second-order primitives of TLA$^+$ are typically specified by an axiom schema, in which case the higher-order arguments are used to generate the right instance. To take a simple example, consider

the expression $\{n \in Int : n \neq i\}$ where i is bound by a quantifier. Internally, we represent this expression as a second-order application $setst(Int, \lambda n : n \neq i)$. To make it first-order, we rewrite it as $setst^\bullet(Int, i)$, where the new operator $setst^\bullet : \iota \times \iota \to \iota$ is specified by

$$\forall i^\iota, s^\iota, n^\iota : mem(n, setst^\bullet(s, i)) \Leftrightarrow mem(n, s) \wedge n \neq i$$

Second-order applications where the operator is not a TLA$^+$ primitive are rewritten in the same way—there is just no axiom schema to instantiate for them.

This method of reduction to first-order logic is simplistic but allows basic reasoning about the second-order TLA$^+$ constructs—set comprehension, set refinement, choose-expressions and explicit functions. Its major flaw is that expressions may come out harder to unify after rewriting. For instance, the simple goal

$$\exists i : \{n \in Int : n \neq i\} = \{n \in Int : n \neq 0\}$$

results in a problem only provable using set extensionality, because the second set is rewritten as $setst^{\bullet\bullet}(Int)$ where $setst^{\bullet\bullet}$ is specified by another instance of the comprehension schema. We attempt to detect when a previously introduced operator can be reused for a rewriting, but our implementation is far from complete.

Heuristics for Set Extensionality It is difficult for SMT solvers to find relevant instances for the axiom of set extensionality, and there is no obvious trigger for it. While some proofs may depend on the axiom of extensionality, they tend to do so in predictable ways. Our support for set extensionality is very limited, but it is implemented easily and suffices for many cases.

The idea is simply to use a special predicate for the sole purpose of triggering the axiom of set extensionality:

$$appext : \iota \times \iota \to o$$
$$\forall x^\iota, y^\iota : \{appext(x, y)\}\ (\forall z^\iota : mem(z, x) \Leftrightarrow mem(z, y)) \Rightarrow x = y$$

Note that only one implication is specified by the axiom—the other implication is trivial and not useful for proofs. It remains to find how relevant instances of $appext(x, y)$ can be generated.

In most obligations where set extensionality is needed, the relevant equality occurs explicitly in the obligation. For these, it would suffice to generate a term $appext(x, y)$ for every $x = y$ in the problem. However, while it is true that every object is a set in TLA$^+$, attempting to prove a goal like $1 + 1 = 2$ by set extensionality would be clearly misguided. Our heuristic is to consider only the equalities where at least one member has a set-theoretic top connective. We also ignore equalities that occur in negative Boolean context, like in $x = \emptyset \Rightarrow y \in x$, as these equalities can be simplified.

The second problem is that the builtin symbol $=$ cannot be used in a trigger. We circumvent this problem by declaring and defining an equivalent rela-

tion *equals*.

$$
\begin{aligned}
&\mathit{equals} : \iota \times \iota \to o \\
&\forall x^\iota, y^\iota : \{\mathit{equals}(x, y)\}\ \ \mathit{equals}(x, y) \Leftrightarrow x = y \\
&\forall x^\iota, y^\iota : \{\mathit{equals}(x, y)\}\ \ \mathit{appext}(x, y)
\end{aligned}
$$

We rewrite the relevant equalities with *equals* for the translation. For example, a goal $a = b \Rightarrow (a \cap c) = (c \cap b)$ is encoded as $a = b \Rightarrow \mathit{equals}(\mathit{cap}(a, c), \mathit{cap}(c, b))$. Set extensionality must only be applied for the second equality. The use of *equals* triggers a match for the two axioms above; the term $\mathit{appext}(\mathit{cap}(a, c), \mathit{cap}(c, b))$ is generated, triggering the axiom.

This technique essentially implements set extensionality as a rewriting rule. In other situations, the relevant instance of extensionality is obvious to the user, but not explicit in the proof. A common situation involves checking that two sets S and T are disjoint, which is expressed $S \cap T = \emptyset$. We can automatize these checks by adding the following axiom to the SMT problem:

$$\forall x^\iota, y^\iota : \{\mathit{cap}(x, y)\}\ \ \mathit{appext}(\mathit{cap}(x, y), \mathit{empty})$$

4 Evaluation

Our SMT encoding is implemented in TLAPS and available on GitHub.[4] We now present its evaluation. The main purpose of this evaluation is to compare our encoding with the original SMT backend.

4.1 Experiment and Results

Our starting data is a collection of TLA$^+$ specifications, taken from three different sources: the library of TLA$^+$ examples,[5] the library of examples from the TLAPS distribution, and a recent specification of Lamport's Deconstructed Bakery algorithm [9]. We did not evaluate TLAPS on the specifications themselves, but instead used it to generate SMT benchmarks, and then evaluate SMT solvers on those benchmarks. For every specification, two SMT benchmarks were generated, one using the old encoding, the other using our version.[6]

We used the following SMT solvers for the evaluation: CVC4, cvc5, Z3, veriT. For veriT, we modify the input file by replacing the SMT logic UFNIA by UFLIA, as veriT only supports linear arithmetic. All solvers are called with a timeout of 5 seconds, which is the default timeout in TLAPS. The experiment was carried out on a Dell Latitude laptop with an Intel Core i7 processor at 1.90 GHz. The results, presented in table 1, show how many obligations were solved using each version of the encoding (top numbers). An obligation is considered solved

[4] https://github.com/tlaplus/tlapm

[5] https://github.com/tlaplus/Examples

[6] The TLA$^+$ specifications and SMT benchmarks generated from them can be found at https://github.com/adef-inr/SafeTLAEncodingBenchmarks

if it is solved by at least one solver. We also computed the numbers of uniquely solved obligations (bottom numbers). An obligation is solved uniquely with one encoding if it is solved while the alternate encoded version is not solved.

Specification	Size	Old	New
TLA$^+$ Examples	1371	1142 35	1265 158
TLAPS Examples	666	583 16	589 22
Deconstructed Bakery	777	652 14	754 116
Total	2814	2377 65	2608 296

Table 1. Obligations solved using the two SMT encodings

4.2 Discussion

Our encoding performed better than the previous one; we solved 92.6% of all obligations with our version against 84.8% with the old version. Our encoding solves 296 obligations that were unsolved before, but 65 obligations are not solved anymore. Note that, for the TLA$^+$ and TLAPS examples, all obligations were originally solved, but not necessarily by the SMT backend. Many proof steps made explicit calls to Zenon or Isabelle. We replaced them by calls to SMT, so our benchmarks contain SMT problems that were not originally solved, which is why the old encoding does not solve everything.

We should remark on the distribution of uniquely solved obligations, which is not shown precisely in the table. For *individual specifications* in the TLA$^+$ and TLAPS examples, those numbers are very low for both encodings. For very small files, each encoding solves about 0–4 obligations uniquely; for larger files, that number is about 5–8. The only exception is the specification Tencent Paxos, which includes a file on which our encoding solved 132 obligations uniquely. This anomaly appears to be the result of a bug in the old encoding, as it fails to produce an output for many obligations. Thus, we may want to account for this anomaly by ignoring Tencent Paxos, in which case the performances of both encodings are actually similar on the TLA$^+$ examples.

The original files from Deconstructed Bakery contain 130 explicit calls to Zenon or Isabelle. The vast majority of the 116 obligations solved uniquely by our encoding come from this set. It is hard to determine the exact reasons for this success. The Deconstructed Bakery specification makes an especially advanced

use of TLA$^+$ functions as it involves partial functions and matrices. Sets of partial functions, for instance, are defined by

$$PFunc(X, Y) \triangleq \text{UNION } \{[XX \to Y] : XX \in \text{SUBSET } X\}$$

The old encoding would rewrite any formula $f \in PFunc(X, Y)$ into a formula containing three quantifiers with no triggers. Our solution does not have that problem, which may be the reason behind its better performances.

Our concern for now is to find explanations for the 65 obligations we do not solve anymore. We are aware of several areas of improvement. Notably, our reduction of second-order applications to first-order could be improved to reuse symbols more often. We are also investigating alternative formulations of the theory of TLA$^+$ functions; our current axioms do not always infer all the relevant facts $f \in [S \to T]$, which may hinder progress on Deconstructed Bakery in particular.

5 Conclusion

We presented an encoding of TLA$^+$'s constant fragment into SMT-LIB. Our approach is based on the view that TLA$^+$ is a standard theory on top of a core logic without formulas. Proof obligations are encoded into SMT's logic by first applying a simple transformation to recover formulas, then inserting declarations and axioms for all relevant TLA$^+$ primitives. We contrast this approach with the original SMT encoding, which attempts to simplify away the TLA$^+$ primitives, but must rely on heavy preprocessing techniques to do so.

Our encoding faithfully translates expressions of TLA$^+$'s untyped set theory. It is easy to implement, therefore safer to use. We used SMT triggers to optimize our axiomatization. To our surprise, we were able to achieve performances similar to the previous version of the encoding with this technique. This runs counter to the idea that TLA$^+$ obligations must be preprocessed and simplified for SMT. Solvers can handle the problems of TLA$^+$ despite the absence of types, because most obligations only require elementary inferences on already explicit sets and functions, and triggers can model these inferences.

Acknowledgment I thank Jasmin Blanchette, Pascal Fontaine, and Stephan Merz for their support and guidance through the development of this work. This research is funded by the European Research Council (ERC) under the European's Union Horizon 2020 research and innovation program (grant agreement No. 713999, Matryoshka), and by the Région Grand Est.

References

1. Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh

Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011.

2. Richard Bonichon, David Delahaye, and Damien Doligez. Zenon : An extensible automated theorem prover producing checkable proofs. In Nachum Dershowitz and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 14th International Conference, LPAR 2007, Yerevan, Armenia, October 15-19, 2007, Proceedings*, volume 4790 of *Lecture Notes in Computer Science*, pages 151–165. Springer, 2007.
3. Thomas Bouton, Diego Caminha Barbosa De Oliveira, David Déharbe, and Pascal Fontaine. veriT: An open, trustable and efficient smt-solver. In Renate A. Schmidt, editor, *Automated Deduction - CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings*, volume 5663 of *Lecture Notes in Computer Science*, pages 151–156. Springer, 2009.
4. Denis Cousineau, Damien Doligez, Leslie Lamport, Stephan Merz, Daniel Ricketts, and Hernán Vanzetto. TLA$^+$ proofs. *CoRR*, abs/1208.5933, 2012.
5. Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, pages 337–340, 2008.
6. Claire Dross, Sylvain Conchon, Johannes Kanig, and Andrei Paskevich. Reasoning with triggers. In *10th International Workshop on Satisfiability Modulo Theories, SMT 2012, Manchester, UK, June 30 - July 1, 2012*, pages 22–31, 2012.
7. Leslie Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, 1994.
8. Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.
9. Leslie Lamport. Deconstructing the bakery to build a distributed state machine. *Commun. ACM*, 65(9):58–66, 2022.
10. Leslie Lamport and Lawrence C. Paulson. Should your specification language be typed. *ACM Trans. Program. Lang. Syst.*, 21(3):502–526, 1999.
11. K. Rustan M. Leino and Clément Pit-Claudel. Trigger selection strategies to stabilize program verifiers. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*, pages 361–381, 2016.
12. Stephan Merz and Hernán Vanzetto. Encoding TLA$^+$ into unsorted and many-sorted first-order logic. *Sci. Comput. Program.*, 158:3–20, 2018.
13. Michal Moskal. Programming with triggers. *ACM International Conference Proceeding Series*, 01 2009.
14. Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How Amazon web services uses formal methods. *Commun. ACM*, 58(4):66–73, 2015.
15. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.
16. Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking TLA$^+$ specifications. In *Correct Hardware Design and Verification Methods, 10th IFIP WG 10.5 Advanced Research Working Conference, CHARME '99, Bad Herrenalb, Germany, September 27-29, 1999, Proceedings*, pages 54–66, 1999.

A Semantics of TLA$^+$ and Proof of Correctness for $\mathcal{B}^o$

We provide details on the semantics of TLA$^+$'s Boolean connectives (Section 2) and a fuller proof of correctness for our transformation $\mathcal{B}^o$ (Section 3.2).

Definition 3 (Interpretations). *A TLA$^+$ domain is a collection D that contains at least two distinct values noted $\top^D$ and $\bot^D$. We associate a collection D_τ to every type τ in the expected way.*

Let Σ be a TLA$^+$ signature. A Σ-interpretation $\mathcal{I}$ consists of a TLA$^+$ domain and a mapping $k \mapsto k^\mathcal{I}$ from the symbols of Σ such that every $k^\mathcal{I}$ is an element of $D_{\Sigma(k)}$. A valuation is a function of variable symbols to elements of D. For all valuations θ, variable x and element v of D, we note θ^x_v the valuation that reassigns x to v.

Given an interpretation $\mathcal{I}$ and a valuation θ, the interpretation of expressions and arguments is defined recursively:

$$\begin{aligned}
[\![x]\!]^\mathcal{I}_\theta &\triangleq \theta(x)\\
[\![k(f_1,\ldots,f_n)]\!]^\mathcal{I}_\theta &\triangleq k^\mathcal{I}([\![f_1]\!]^\mathcal{I}_\theta,\ldots,[\![f_n]\!]^\mathcal{I}_\theta)\\
[\![e_1 = e_2]\!]^\mathcal{I}_\theta &\triangleq \top^D \text{ if } [\![e_1]\!]^\mathcal{I}_\theta = [\![e_2]\!]^\mathcal{I}_\theta, \text{ otherwise } \bot^D\\
[\![\text{FALSE}]\!]^\mathcal{I}_\theta &\triangleq \bot^D\\
[\![e_1 \Rightarrow e_2]\!]^\mathcal{I}_\theta &\triangleq \top^D \text{ if } [\![e_1]\!]^\mathcal{I}_\theta \neq \top^D \text{ or } [\![e_2]\!]^\mathcal{I}_\theta = \top^D, \text{ otherwise } \bot^D\\
[\![\forall x : e]\!]^\mathcal{I}_\theta &\triangleq \top^D \text{ if } [\![e]\!]^\mathcal{I}_{\theta^x_v} = \top^D \text{ for all } v \text{ in } D, \text{ otherwise } \bot^D
\end{aligned}$$

For all $v_1,\ldots,v_n$ in D^n,

$$\begin{aligned}
[\![k]\!]^\mathcal{I}_\theta\,(v_1,\ldots,v_n) &\triangleq k^\mathcal{I}(v_1,\ldots,v_n)\\
[\![\lambda x_1,\ldots,x_n : e]\!]^\mathcal{I}_\theta\,(v_1,\ldots,v_n) &\triangleq [\![e]\!]^\mathcal{I}_{\theta^{x_1,\ldots,x_n}_{v_1,\ldots,v_n}}
\end{aligned}$$

We admit that the valuation θ does not affect the interpretation of expressions with no free variables. This justifies the notations $[\![e]\!]^\mathcal{I}$ and $\mathcal{I} \models e$. Remark that the equation for implication above is not the same as

$$[\![e_1 \Rightarrow e_2]\!]^\mathcal{I}_\theta \triangleq \top^D \text{ if } [\![e_1]\!]^\mathcal{I}_\theta = \bot^D \text{ or } [\![e_2]\!]^\mathcal{I}_\theta = \top^D, \text{ otherwise } \bot^D$$

Indeed, for any value v, $v \neq \top^D$ does not entail $v = \bot^D$. A consequence of our definition is that $c \Rightarrow c$ is a tautology for all constant c.

We now prove our correctness result for $\mathcal{B}^o$:

Theorem 3. *Let $\mathcal{I}$ be a TLA$^+$ interpretation. The following propositions hold for all expressions e, arguments f, and valuations θ:*

i) $[\![\mathcal{B}^\iota(e)]\!]^{\mathcal{I}^\mathcal{B}}_\theta = [\![e]\!]^\mathcal{I}_\theta$
ii) $[\![\mathcal{B}^o(e)]\!]^{\mathcal{I}^\mathcal{B}}_\theta = \top$ *iff* $[\![e]\!]^\mathcal{I}_\theta = \top^D$
iii) $[\![\mathcal{B}^o(e)]\!]^{\mathcal{I}^\mathcal{B}}_\theta = \bot$ *implies* $[\![e]\!]^\mathcal{I}_\theta = \bot^D$ *if* $\mathcal{B}^o(e)$ *is not a projection*
iv) $[\![\mathcal{B}^f(f)]\!]^{\mathcal{I}^\mathcal{B}}_\theta = [\![f]\!]^\mathcal{I}_\theta$

Proof. The proof is by induction on the construction of the result. We treat the cases of injection and projection, and the case of implication. All other cases are either straightforward or analogous.

Injection into Bool. Let $\mathcal{B}^\iota(e) \triangleq cast_o(\mathcal{B}^o(e))$. We must prove property i) for $\mathcal{B}^\iota(e)$. By definition:

$$[\![\mathcal{B}^\iota(e)]\!]_\theta^{\mathcal{I}^\mathcal{B}} = cast_o^{\mathcal{I}^\mathcal{B}}([\![\mathcal{B}^o(e)]\!]_\theta^{\mathcal{I}^\mathcal{B}}) = \begin{cases} \top^D & \text{if } [\![\mathcal{B}^o(e)]\!]_\theta^{\mathcal{I}^\mathcal{B}} = \top \\ \bot^D & \text{otherwise} \end{cases}$$

The induction hypothesis applies to $\mathcal{B}^o(e)$. If $[\![\mathcal{B}^o(e)]\!]_\theta^{\mathcal{I}^\mathcal{B}} = \top$ then $[\![e]\!]_\theta^{\mathcal{I}} = \top^D$ by property ii). If $[\![\mathcal{B}^o(e)]\!]_\theta^{\mathcal{I}^\mathcal{B}} = \bot$, we deduce $[\![e]\!]_\theta^{\mathcal{I}} = \bot^D$ from property iii) and the fact that $\mathcal{B}^o(e)$ cannot be a projection. Indeed, by construction of $\mathcal{B}^\iota$ and $\mathcal{B}^o$, a projection cannot be followed by an injection. In both cases, we have $[\![\mathcal{B}^\iota(e)]\!]_\theta^{\mathcal{I}^\mathcal{B}} = [\![e]\!]_\theta^{\mathcal{I}}$.

Projection onto Bool. Let $\mathcal{B}^o(e) \triangleq \mathcal{B}^\iota(e) = cast_o(\text{TRUE})$. Since we are treating the projection case, the only property we really need to prove is property ii). We have the following equivalences:

$$\begin{aligned} [\![\mathcal{B}^o(e)]\!]_\theta^{\mathcal{I}^\mathcal{B}} = \top \textit{ iff } & [\![\mathcal{B}^\iota(e)]\!]_\theta^{\mathcal{I}^\mathcal{B}} = \top^D && (\text{since } [\![cast_o(\text{TRUE})]\!]_\theta^{\mathcal{I}^\mathcal{B}} = \top^D) \\ \textit{iff } & [\![e]\!]_\theta^{\mathcal{I}} = \top^D && (\text{by property i) on } \mathcal{B}^\iota(e)) \end{aligned}$$

Implication. Let $e \triangleq e_1 \Rightarrow e_2$ and $\mathcal{B}^o(e) \triangleq \mathcal{B}^o(e_1) \Rightarrow \mathcal{B}^o(e_2)$. We must prove properties ii) and iii). But remark that the former implies the latter immediately, as $[\![e]\!]_\theta^{\mathcal{I}} \neq \top^D$ implies $[\![e]\!]_\theta^{\mathcal{I}} = \bot^D$ when e is an implication. Property ii) is proven by the following series of equivalences:

$$\begin{aligned} & [\![\mathcal{B}^o(e_1 \Rightarrow e_2)]\!]_\theta^{\mathcal{I}^\mathcal{B}} = \top \\ & \quad \textit{iff } [\![\mathcal{B}^o(e_1)]\!]_\theta^{\mathcal{I}^\mathcal{B}} = \bot \text{ or } [\![\mathcal{B}^o(e_2)]\!]_\theta^{\mathcal{I}^\mathcal{B}} = \top && (\text{by the usual semantics of } \Rightarrow) \\ & \quad \textit{iff } [\![e_1]\!]_\theta^{\mathcal{I}} \neq \top^D \text{ or } [\![e_2]\!]_\theta^{\mathcal{I}} = \top^D && (\text{by property ii})) \\ & \quad \textit{iff } [\![e_1 \Rightarrow e_2]\!]_\theta^{\mathcal{I}} = \top^D && (\text{by TLA}^+\text{'s semantics of } \Rightarrow) \end{aligned}$$

□