HAL
archives-ouvertes.fr

Encoding TLA+ set theory into many-sorted first-order logic

Stephan Merz, Hernán Vanzetto

To cite this version:

Stephan Merz, Hernán Vanzetto. Encoding TLA+ set theory into many-sorted first-order logic. 2015. <hal-01244627>

HAL Id: hal-01244627

https://hal.inria.fr/hal-01244627

Submitted on 16 Dec 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Encoding TLA$^+$ set theory into many-sorted first-order logic

Stephan Merz and Hernán Vanzetto

Inria, Villers-lès-Nancy, F-54600, France
Université de Lorraine, LORIA, UMR 7503, Vandoeuvre-lès-Nancy, F-54506, France
CNRS, LORIA, UMR 7503, Vandoeuvre-lès-Nancy, F-54506, France

Abstract. We present an encoding of Zermelo-Fraenkel set theory into many-sorted first-order logic, the input language of state-of-the-art SMT solvers. This translation is the main component of a back-end prover based on SMT solvers in the TLA$^+$ Proof System.

1 Introduction

The specification language TLA$^+$ [11] combines a variant of Zermelo-Fraenkel (ZF) set theory for the description of the data manipulated by algorithms, and linear-time temporal logic for the specification of their behavior. The TLA$^+$ Proof System (TLAPS) provides support for mechanized reasoning about TLA$^+$ specifications; it integrates backends for making automatic reasoners available to users of TLAPS. The work reported here is motivated by the development of an SMT backend through which users of TLAPS interact with off-the-shelf SMT (satisfiability modulo theories) solvers for non-temporal reasoning in the set theory of TLA$^+$.

More specifically, TLAPS is built around a so-called Proof Manager that interprets the proofs occurring in the TLA$^+$ module provided by the user, generates corresponding proof obligations, and passes them to external automated verifiers, which are the back-end provers of TLAPS.

Previous to this work, three back-end provers with different capabilities were available: Isabelle/TLA$^+$, a faithful encoding of TLA$^+$ set theory in the Isabelle proof assistant, which provides automated proof methods based on first-order reasoning and rewriting; Zenon [5], a tableau prover for first-order logic with equality that includes extensions for reasoning about sets and functions; and a backend called SimpleArithmetic, now deprecated, implementing a decision procedure for Presburger arithmetic.[1]

[1] The backends available prior to the work presented here also included a generic translation to the input language of SMT solvers that focused on quantifier-free formulas of linear arithmetic. This SMT backend was occasionally useful because the other backends perform quite poorly on obligations involving arithmetic reasoning. However, it covered only a small subset of TLA$^+$.

The Isabelle and Zenon backends have very limited support for arithmetic reasoning, while SimpleArithmetic handles only pure arithmetic formulas, requiring the user to manually decompose the proofs until the corresponding proof obligations fall within the respective fragments. Beyond its integration as a semi-automatic backend, Isabelle/TLA$^+$ serves as the most trusted back-end prover. Accordingly, it is also intended for certifying proof scripts produced by other back-end provers. When possible, backends are expected to produce a detailed proof that can be checked by Isabelle/TLA$^+$. Currently, only the Zenon backend has an option for exporting proofs that can be certified in this way.

In this paper we describe the foundations of a back-end prover based on SMT solvers for non-temporal proof obligations arising in TLAPS.[2] When verifying distributed algorithms, proof obligations are usually "shallow", but they still require many details to be checked: interactive proofs can become quite large without powerful automated back-end provers that can cope with a significant fragment of the language. TLA$^+$ heavily relies on modeling data using sets and functions. Tuples and records, which occur very often in TLA$^+$ specifications, are defined as functions. Assertions mixing first-order logic (FOL) with sets, functions, and arithmetic expressions arise frequently in safety proofs of TLA$^+$ specifications. Accordingly, we do not aim at proofs of deep theorems of mathematical set theory but at good automation for obligations mixing elementary set expressions, functions, records, and (linear) integer arithmetic, and our main focus is on SMT solvers, although we have also used the techniques described here with FOL provers. The de-facto standard input language for SMT solvers is SMT-LIB [3], which is based on multi-sorted FOL (MS-FOL [12]).

In Section 3 we present the translation from TLA$^+$ to MS-FOL. Although some of the encoding techniques that we use can be found in similar tools for other set-theoretic languages, the particularities of TLA$^+$ make the translation non-trivial:

- Since TLA$^+$ is untyped, "silly" expressions such as $3 \cup \text{TRUE}$ are legal; they denote some (unspecified) value. TLA$^+$ does not even distinguish between Boolean and non-Boolean expressions, hence Boolean values can be stored in data structures just like any other values.
- Functions, which are defined axiomatically, are total and have a domain. This means that a function applied to an element of its domain has the expected value but for any other argument, the value of the function application is unspecified. Similarly, the behavior of arithmetic operators is specified only for integer arguments.
- TLA$^+$ is equipped with a deterministic choice operator (Hilbert's ε operator), which has to be soundly encoded.

The first item is particularly challenging for our objectives: whereas an untyped language is very expressive and flexible for writing specifications, standard MS-FOL reasoners rely on types for good automation. In order to support TLA$^+$

[2] Non-temporal reasoning is enough for proving safety properties and makes up the vast majority of proof steps in liveness proofs.

expressions in a many-sorted environment, we use only one sort to encode all TLA$^+$ expressions. We therefore call this translation the "untyped" encoding of TLA$^+$, where type inference of sorted expressions such as arithmetic is essentially delegated to the solvers. In the following we will use the terms *type* and *sort* interchangeably.

Paper outline. Section 2 introduces the set theory of TLA$^+$, Section 3 presents our translation from TLA$^+$ to MS-FOL. Section 4 provides experimental results, and Section 5 concludes and gives directions for future work.

Related work. In previous publications [14, 15], we presented primitive encodings of TLA$^+$ into SMT-LIB, where CHOOSE expressions were not fully supported and Boolification was not made explicit in the translation. As a preprocessing step, we developed a type system with dependent and refinement types for TLA$^+$ [16]: an algorithm takes a TLA$^+$ proof obligation and annotates it with types, which are then used to simplify our encoding [15].

Some of the encoding techniques presented in Section 3 were already defined before or are simply folklore, but to our knowledge they have not been combined and studied in this way. Moreover, the idiosyncrasies of TLA$^+$ render their applicability non-trivial. For instance, TLA$^+$'s axiomatized functions with domains, including tuples and records, are deeply rooted in the language.

The Rodin tool set supporting Event-B is based on two translations. The SMT solvers plugin [6] directly encodes simple sets (*i.e.*, excluding set of sets) as polymorphic λ-expressions, which are non-standard and are only handled by the parser of the veriT SMT solver. The ppTrans plugin [10] generates different SMT sorts for each basic set and every combination of sets (power sets or cartesian products) found in the proof obligation. Similarly, Mentre et al. [13] rely on Why3 as an interface to discharge Atelier-B proof obligations using different SMT solvers, with sets having a polymorphic type.

Recently, Delahaye et al. [7] proposed a different approach to reason about set theory, instead of a direct encoding into FOL. The theory of deduction modulo is an extension of predicate calculus that includes rewriting of terms and of propositions, and which is well suited for proof search in axiomatic theories, as it turns axioms into rewrite rules. For example, Peano arithmetic or Zermelo set theory can be encoded without axioms, turning the proof search based on axioms into computations. Zenon Modulo [7, 9] implements deduction modulo within a first-order theorem prover.

MPTP [17] translates Mizar to TPTP/FOF. The Mizar language has second-order predicate variables and abstract terms derived from replacement and comprehension, such as the set $\{n - m \text{ where } m, n \text{ is } \mathit{Integer} : n < m\}$. During preprocessing, MPTP replaces them by fresh symbols, with their definitions at the top level. Similar to our abstraction technique (cf. Section 3.3.3), it is comparable to Skolemization. In contrast to our intended application, MPTP is mainly targeted at mathematical reasoning.

2 TLA$^+$ set theory

In this section we describe a fragment of the language of proof obligations generated by the TLA$^+$ Proof System that is relevant for this paper. This language is a variant of FOL with equality, extended in particular by syntax for set, function and arithmetic expressions, and a construct for a deterministic choice operator. For a complete presentation of the TLA$^+$ language see [11, Sec. 16].

We assume given two non-empty, infinite, and disjoint collections $\mathcal{V}$ of *variable* symbols, and $\mathcal{O}$ of *operator* symbols,[3] each equipped with its arity. The only syntactical category in the language is the *expression*. For presentational purposes we distinguish between terms, formulas, set objects, etc. An expression e is inductively defined by the following grammar:

$$
\begin{array}{rll}
e ::= & v \mid w(e, \ldots, e) & \text{(terms)} \\
& \mid \textsc{false} \mid e \Rightarrow e \mid \forall v\colon\ e \mid e = e \mid e \in e & \text{(formulas)} \\
& \mid \{\} \mid \{e, e\} \mid \textsc{subset}\ s \mid \textsc{union}\ s \mid \{v \in e : e\} \mid \{e : v \in e\} & \text{(sets)} \\
& \mid \textsc{choose}\ x\colon e & \text{(choice)} \\
& \mid e[e] \mid \textsc{domain}\ e \mid [v \in e \mapsto e] \mid [e \to e] & \text{(functions)} \\
& \mid 0 \mid 1 \mid 2 \mid \ldots \mid \mathit{Int} \mid \mathit{Nat} \mid -e \mid e + e \mid e < e \mid e \mathbin{..} e & \text{(arithmetic)} \\
& \mid \textsc{if}\ e\ \textsc{then}\ e\ \textsc{else}\ e & \text{(conditional)}
\end{array}
$$

A *term* is a variable symbol v in $\mathcal{V}$ or an application of an operator symbol w in $\mathcal{O}$ to expressions. *Formulas* are built from FALSE, implication and universal quantification, and from the binary operators $=$ and $\in$. From these formulas, we can define the familiar constant TRUE, the unary $\neg$ and the binary connectives $\wedge$, $\vee$, $\Leftrightarrow$, and the existential quantifier $\exists$. Also, $\forall x \in S\colon e$ is defined as $\forall x\colon x \in S \Rightarrow e$. In standard set theory, sets are constructed from axioms that state their existence. TLA$^+$ has explicit syntax for set objects (empty set, pairing, power set, generalized union, and two forms of set comprehension derived from the standard axiom schema of replacement), whose semantics is defined axiomatically. Since TLA$^+$ is a set theoretic language, every expression – including formulas, functions, numbers, etc. – denotes a set.

Another primitive construct of TLA$^+$ is Hilbert's choice operator ε, written CHOOSE $x\colon P(x)$, that denotes an arbitrary but fixed value x such that $P(x)$ is true, provided that such a value exists. Otherwise the value of CHOOSE $x\colon P(x)$ is arbitrary. The semantics of CHOOSE is expressed by the following axiom schemas. The first one gives an alternative way of defining quantifiers, and the second one expresses that CHOOSE is deterministic.

$$\big(\exists x\colon P(x)\big) \ \Leftrightarrow\ P\big(\textsc{choose}\ x\colon P(x)\big) \tag{1}$$

$$\big(\forall x\colon P(x) \Leftrightarrow Q(x)\big) \ \Rightarrow\ \big(\textsc{choose}\ x\colon P(x)\big) = \big(\textsc{choose}\ x\colon Q(x)\big) \tag{2}$$

From axiom (2) note that if there is no value satisfying some predicate P, *i.e.*, $\forall x\colon P(x) \Leftrightarrow \textsc{false}$ holds, then $(\textsc{choose}\ x\colon P(x)) = (\textsc{choose}\ x\colon \textsc{false})$. Con-

[3] TLA$^+$ operator symbols correspond to the standard function and predicate symbols of first-order logic but we reserve the term "function" for TLA$^+$ functional values.

sequently, the expression CHOOSE x : FALSE and all its equivalent forms represent a unique value.

Certain TLA$^+$ values are *functions*. Unlike standard ZF set theory, TLA$^+$ functions are not defined as sets of pairs, but TLA$^+$ provides primitive syntax associated with functions. The expression $f[e]$ denotes the result of applying function f to e, DOMAIN f denotes the domain of f, and $[x \in S \mapsto e]$ denotes the function g with domain S such that $g[x] = e$, for any $x \in S$. For $x \notin S$, the value of $g[x]$ is unspecified. A TLA$^+$ value f is a function if and only if it satisfies the predicate $IsAFcn(f)$ defined as $f = [x \in \text{DOMAIN } f \mapsto f[x]]$. The fundamental law governing TLA$^+$ functions is

$$f = [x \in S \mapsto e] \Leftrightarrow IsAFcn(f) \wedge \text{DOMAIN } f = S \wedge \forall x \in S : f[x] = e \qquad (3)$$

Natural numbers $0, 1, 2, \ldots$ are primitive symbols of TLA$^+$. Standard modules of TLA$^+$ define *Int* to denote the set of integer numbers, the operators $+$ and $<$ are interpreted in the standard way when their arguments are integers, and the interval $a \,..\, b$ is defined as $\{n \in Int : a \leq n \wedge n \leq b\}$.

3 Untyped encoding of TLA$^+$ into MS-FOL

We define a translation from TLA$^+$ to multi-sorted first-order logic. Given a TLA$^+$ proof obligation, the system generates an equi-satisfiable formula whose proof can be attempted with automatic theorem provers, including SMT solvers.

The translation proceeds in two main steps. First, a preprocessing and optimization phase applies satisfiability-preserving transformations to a given TLA$^+$ formula in order to remove expressions that the target solver cannot handle. The result is an intermediate *basic* TLA$^+$ formula, *i.e.*, a TLA$^+$ expressions that has an obvious counterpart in the SMT-LIB/AUFLIA language. A basic TLA$^+$ formula is composed only of TLA$^+$ terms and formulas, including equality and set membership relations, plus primitive arithmetic operators and IF-THEN-ELSE expressions. All expressions having a truth value are mapped to the sort Bool, and we declare a new sort U (for TLA$^+$ universe) for all non-Boolean expressions, including sets, functions, and numbers. Thus, we call this the *untyped* encoding.

3.1 Boolification

Since TLA$^+$ has no syntactic distinction between Boolean and non-Boolean expressions, we first need to determine which expressions are used as propositions. We adopt the liberal interpretation of TLA$^+$ Boolean expressions where any expression with a top-level connective among logical operators, $=$, and $\in$ has a Boolean value.[4] Moreover, the result of any expression with a top-level logical

[4] The standard semantics of TLA$^+$ offers three alternatives to interpret expressions [11, Sec. 16.1.3]. In the liberal interpretation, an expression like $42 \Rightarrow \{\}$ always has a truth value, but it is not specified if that value is true or false. In the conservative and moderate interpretations, the value of $42 \Rightarrow \{\}$ is completely unspecified.

connective agrees with the result of the expression obtained by replacing every argument e of that connective with $(e = \text{TRUE})$.

For example, consider the expression $\forall x : (\neg\neg x) = x$, which is not a theorem. Indeed, x need not be Boolean, whereas $\neg\neg x$ is necessarily Boolean, hence we may not conclude that the expression is valid. However, $\forall x : (\neg\neg x) \Leftrightarrow x$ is valid because it is interpreted as $\forall x : (\neg\neg(x = \text{TRUE})) \Leftrightarrow (x = \text{TRUE})$. Observe that the value of $x = \text{TRUE}$ is a Boolean for any x, although the value is unspecified if x is non-Boolean.

In order to identify the expressions used as propositions we use a simple algorithm that recursively traverses an expression searching for sub-expressions that should be treated as formulas. Expressions e that are used as Booleans, *i.e.*, that could equivalently be replaced by $e = \text{TRUE}$, are marked as e^b, whose definition can be thought of as $e^b \triangleq e = \text{TRUE}$. This only applies if e is a term, a function application, or a CHOOSE expression. If an expression which is known to be non-Boolean by its syntax, such as a set or a function, is attempted to be Boolified, meaning that a formula is expected in its place, the algorithm aborts with a "type" error. In SMT-LIB we encode x^b as $\mathsf{boolify}(x)$, with $\mathsf{boolify} : \mathsf{U} \to \mathsf{Bool}$. The above examples are translated as $\forall x^{\mathsf{U}} : (\neg\neg\mathsf{boolify}(x)) = x$ and $\forall x^{\mathsf{Bool}} : (\neg\neg x) \Leftrightarrow x$ and their (in)validity becomes evident.

3.2 Direct embedding

Our encoding maps in an almost verbatim way Boolified TLA$^+$ expressions to corresponding formulas in the target language, without changing substantially the structure of the original formula. The goal is to encode TLA$^+$ expressions using essentially first-order logic and uninterpreted functions. For first-order TLA$^+$ expressions it suffices to apply a shallow embedding into the target language. Nonlogical TLA$^+$ operators are declared as function or predicate symbols with U-sorted arguments. For example, the operators $\cup$ and $\in$ are encoded in SMT-LIB as the functions $\mathsf{union} : \mathsf{U} \times \mathsf{U} \to \mathsf{U}$ and $\mathsf{in} : \mathsf{U} \times \mathsf{U} \to \mathsf{Bool}$.

The semantics of standard TLA$^+$ operators are defined axiomatically. The only primitive set-theoretical operator is $\in$, so the function in will remain unspecified, while we can express in MS-FOL the axiom for $\cup$ as

$$\forall x^{\mathsf{U}}, S^{\mathsf{U}}, T^{\mathsf{U}}.\ \ \mathsf{in}(x, \mathsf{union}(S, T)) \Leftrightarrow \mathsf{in}(x, S) \vee \mathsf{in}(x, T) \tag{4}$$

Note that sets are just values in the universe of discourse (represented by the sort U in the sorted translation), and it is therefore possible to represent sets of sets and to quantify over sets. The construct for set enumeration $\{e_1, \ldots, e_n\}$, with $n \geq 0$, is an n-ary expression, so we declare separate uninterpreted functions for the arities that occur in the proof obligation, together with the corresponding axioms.

Only in the moderate and liberal interpretation, the expression $\text{FALSE} \Rightarrow \{\}$ has a Boolean value, and that value is true. In the liberal interpretation, all the ordinary laws of logic, such as commutativity of $\wedge$, are valid, even for non-Boolean arguments.

In order to reason about the theory of arithmetic, an automated prover requires type information, either generated internally, or provided explicitly in the input language. The axioms that we have presented so far rely on FOL over uninterpreted function symbols over the single sort U. For arithmetic reasoning, we want to benefit from the prover's native capabilities. We declare an unspecified, injective function $\mathsf{i2u} : \mathsf{Int} \to \mathsf{U}$ that embeds built-in integers into the sort U. The typical injectivity axiom $\forall m^{\mathsf{Int}}, n^{\mathsf{Int}} : \mathsf{i2u}(m) = \mathsf{i2u}(n) \Rightarrow m = n$ generates instantiation patterns for every pair of occurrences of $\mathsf{i2u}$. Noting that $\mathsf{i2u}$ is injective iff it has a partial inverse, we use instead the axiom $\forall n^{\mathsf{Int}} : \mathsf{u2i}(\mathsf{i2u}(n)) = n$, which generates a linear number of $\mathsf{i2u}(n)$ instances, where the inverse $\mathsf{u2i} : \mathsf{U} \to \mathsf{Int}$ is unspecified. Integer literals k are encoded as $\mathsf{i2u}(k)$.

For example, the formula $3 \in \mathit{Int}$ is translated as $\mathsf{in}(\mathsf{i2u}(3), \mathsf{tla_Int})$ and we have to add to the translation the axiom for Int:

$$\forall x^{\mathsf{U}} : \mathsf{in}(x, \mathsf{tla_Int}) \Leftrightarrow \exists n^{\mathsf{Int}} : x = \mathsf{i2u}(n) \tag{5}$$

Observe that this axiom introduces two quantifiers to the translation. We can avoid the universal quantifier by encoding expressions of the form $x \in \mathit{Int}$ directly into $\exists n^{\mathsf{Int}} : x = \mathsf{i2u}(n)$, but the provers would still have to deal with the existential quantifier.

Arithmetic operators over TLA$^+$ values are defined homomorphically over the image of $\mathsf{i2u}$ by axioms such as

$$\forall m^{\mathsf{Int}}, n^{\mathsf{Int}} : \mathsf{plus}(\mathsf{i2u}(m), \mathsf{i2u}(n)) \ = \ \mathsf{i2u}(m + n) \tag{6}$$

where $+$ denotes the built-in addition over integers. For other arithmetic operators we define analogous axioms.

In all these cases, type inference is, in some sense, delegated to the back-end prover. The link between built-in operations and their TLA$^+$ counterparts is effectively defined only for values in the range of the function $\mathsf{i2u}$. This approach can be extended to other useful theories that are natively supported, such as arrays or algebraic datatypes.

3.3 Preprocessing and optimizations

The above encoding has two limitations. First, some TLA$^+$ expressions cannot be written in first-order logic. Namely, they are $\{x \in S : P\}$, $\{e : x \in S\}$, CHOOSE $x : P$, and $[x \in S \mapsto e]$, where the predicate P and the expression e, both of which may have x as free variable, become second-order variables when quantified. Secondly, the above encoding does not perform and scale well in practice. State-of-the-art SMT solvers provide *instantiation patterns* to control the potential explosion in the number of ground terms generated for instantiating quantified variables, but we have not been able to come up with patterns to attach to the axiom formulas that would significantly improve the performance, even for simple theorems.

What we do instead is to perform several transformations to the TLA$^+$ proof obligation to obtain an equi-satisfiable formula which can be straightforwardly passed to the solvers using the above encoding.

3.3.1 Normalization. We define a rewriting process that systematically expands definitions of non-basic operators. Instead of letting the solver find instances of the background axioms, it applies the "obvious" instances of those axioms during the translation. In most cases, we can eliminate all non-basic operators. For instance, the ZF axiom for the UNION operator yields the rewriting rule $x \in \text{UNION } S \longrightarrow \exists T \in S : x \in T$.

All defined rewriting rules apply equivalence-preserving transformations. We ensure soundness by proving in Isabelle/TLA$^+$ that all rewriting rules correspond to theorems of TLA$^+$. The theorem corresponding to a rule $e \longrightarrow f$ is $\forall \mathbf{x} : e \Leftrightarrow f$ when e and f are Boolean expressions and $\forall \mathbf{x} : e = f$ otherwise, where $\mathbf{x}$ denotes all free variables in the rule. Most of these theorems exist already in the standard library of Isabelle/TLA$^+$'s library.

The standard extensionality axiom for sets is unwieldy because it introduces an unbounded quantifier, which can be instantiated by any value of sort U. We therefore decided not to include it in the default background theory. Instead, we instantiate equality expressions $x = y$ whenever possible with the extensionality property corresponding to x or y. In these cases, we say that we *expand* equality. For each set expression T we derive rewriting rules for equations $x = T$ and $T = x$. For instance, the rule $x = \{z \in S : P\} \longrightarrow \forall z : z \in x \Leftrightarrow z \in S \wedge P$ is derived from set extensionality and the ZF axiom of bounded set comprehension.

By not including general extensionality, the translation becomes incomplete. Even if we assume that the automated theorem provers are semantically complete, it may happen that the translation of a semantically valid TLA$^+$ formula becomes invalid when encoded. In these cases, the user will need to explicitly add the axiom to the TLA$^+$ proof.

We also include the rule $\forall z : z \in x \Leftrightarrow z \in y \longrightarrow x = y$ for the *contraction* of set extensionality, which we apply with higher priority than the expansion rules. All above rules of the form $\phi \longrightarrow \psi$ define a term rewriting system [2] noted $(\text{TLA}^+, \longrightarrow)$, where $\longrightarrow$ is a binary relation over well-formed TLA$^+$ expressions.

Theorem 1. $(\text{TLA}^+, \longrightarrow)$ *terminates and is confluent.*

Proof (idea). Termination is proved by embedding $(\text{TLA}^+, \longrightarrow)$ into another reduction system that is known to terminate, typically $(\mathbb{N}, >)$ [2]. The embedding is through an ad-hoc monotone mapping μ such that $\mu(a) > \mu(b)$ for every rule $a \longrightarrow b$. It is defined in such a way that every rule instance strictly decreases the number of non-basic and complex expressions such as quantifiers or arithmetic expressions. For confluence, by Newman's lemma [2], it suffices to prove that all critical pairs are joinable. Thus, we just need to find the critical pairs $\langle e_1, e_2 \rangle$ between all combinations of rewriting rules, and then prove that e_1 and e_2 are joinable for each such pair. In particular, the contraction rule is necessary to obtain a strong normalizing system. □

3.3.2 Functions. A TLA$^+$ function $[x \in S \mapsto e(x)]$ is akin to a "bounded" λ-abstraction: the function application $[x \in S \mapsto e(x)][y]$ reduces to the expected

value $e(y)$ if the argument y is an element of S, as stated by the axiom (3). As a consequence, e.g., the formula

$$f = [x \in \{1,2,3\} \mapsto x * x] \;\Rightarrow\; f[0] < f[0] + 1,$$

although syntactically well-formed, should not be provable. Indeed, since 0 is not in the domain of f, we cannot even deduce that $f[0]$ is an integer.

We represent the application of an expression f to another x by two distinct first-order terms depending on whether the *domain condition* $x \in \textsc{domain}\ f$ holds or not: we introduce binary operators α and ω with conditional definitions $x \in \textsc{domain}\ f \;\Rightarrow\; \alpha(f,x) = f[x]$ and $x \notin \textsc{domain}\ f \;\Rightarrow\; \omega(f,x) = f[x]$. From these definitions, we can derive the theorem

$$f[x] = \textsc{if}\ x \in \textsc{domain}\ f\ \textsc{then}\ \alpha(f,x)\ \textsc{else}\ \omega(f,x) \tag{7}$$

that gives a new defining equation for function application. In this way, functions are just expressions that are conditionally related to their argument by α and ω.

The expression $f[0]$ in the above example is encoded as

$$\textsc{if}\ 0 \in \textsc{domain}\ f\ \textsc{then}\ \alpha(f,0)\ \textsc{else}\ \omega(f,0).$$

The solver would have to use the hypothesis to deduce that $\textsc{domain}\ f = \{1,2,3\}$, reducing the condition $0 \in \textsc{domain}\ f$ to false. The conclusion can then be simplified to $\omega(f,0) < \omega(f,0) + 1$, which cannot be proved, as expected. Another example is $f[x] = f[y]$ in a context where $x = y$ holds: the formula is valid irrespective of whether the domain conditions hold or not.

Whenever possible, we try to avoid the encoding of function application as in the definition (7). From (3) and (7), we deduce the rewriting rule:

$$[x \in S \mapsto e][a] \longrightarrow \textsc{if}\ a \in S\ \textsc{then}\ e[x \leftarrow a]\ \textsc{else}\ \omega([x \in S \mapsto e], a) \tag{8}$$

where $e[x \leftarrow a]$ denotes e with a substituted for x. These rules replace two non-basic operators (function application and the function expression) in the left-hand side by only one non-basic operator in the right-hand side (the first argument of ω).

The expression $[x \in S \mapsto e]$ cannot be mapped directly to a first-order expression. Even in sorted languages like MS-FOL, functions have no notion of function domain other than the types of their arguments. Explicit functions will be treated by the abstraction method below. What we can do for the moment is to expand equalities involving functions. The following rewriting rule derived from axiom (3) replaces the function construct by a formula containing only basic operators:

$$f = [x \in S \mapsto e] \longrightarrow \mathit{IsAFcn}(f) \wedge \textsc{domain}\ f = S \wedge \forall x \in S\colon \alpha(f,x) = e$$

Observe that we have simplified $f[x]$ by $\alpha(f,x)$, because $x \in \textsc{domain}\ f$. This mechanism summarizes the essence of the abstraction method to deal with non-basic operators described in the next subsection.

In order to prove that two functions are equal, we need to add a background axiom that expresses the extensionality property for functions:

$$\begin{aligned}\forall f, g\colon\ & \wedge\ \mathit{IsAFcn}(f) \wedge \mathit{IsAFcn}(g)\\ & \wedge\ \textsc{domain}\ f = \textsc{domain}\ g\\ & \wedge\ \forall x \in \textsc{domain}\ g\colon \alpha(f, x) = \alpha(g, x)\\ & \Rightarrow f = g\end{aligned}$$

Again, note that $f[x]$ and $g[x]$ were simplified using α. Unlike set extensionality, this formula is guarded by IsAFcn, avoiding the instantiation of expressions that are not considered functions. To prove that $\textsc{domain}\ f = \textsc{domain}\ g$, we still need to add to the translation the set extensionality axiom, which we abstain from. Instead, reasoning about the equality of domains can be solved with an instance of set extensionality for DOMAIN expressions only.

TLA$^+$ defines n-tuples as functions with domain $1..n$ and records as functions whose domain is a finite set of strings. By treating them as non-basic expressions, we just need to add suitable rewriting rules to $(\text{TLA}^+, \longrightarrow)$, in particular those for extensionality expansion.

3.3.3 Abstraction. Applying rewriting rules does not always suffice for obtaining formulas in basic normal form. As a toy example, consider the valid proof obligation $\forall x\colon P(\{x\} \cup \{x\}) \Leftrightarrow P(\{x\})$. The impediment is that the non-basic sub-expressions $\{x\} \cup \{x\}$ and $\{x\}$ do not occur in the form expected by the left-hand sides of rewriting rules. They must first be transformed into a form suitable for rewriting.

We call the technique described here *abstraction* of non-basic expressions. After applying rewriting, some non-basic expression ψ may remain in the proof obligation. For every occurrence of ψ, we introduce in its place a fresh term y, and add the formula $y = \psi$ as an assumption in the appropriate context. The new term acts as an *abbreviation* for the non-basic expression, and the equality acts as its *definition*, paving the way for a transformation to a basic expression using the above rewriting rules. Non-basic expressions occurring more than once are replaced by the same fresh symbol.

In our example, the expressions $\{x\} \cup \{x\}$ and $\{x\}$ are replaced by fresh constant symbols $k_1(x)$ and $k_2(x)$. Then, the abstracted formula is

$$\begin{aligned}& \wedge\ \forall y_1\colon k_1(y_1) = \{y_1\} \cup \{y_1\}\\ & \wedge\ \forall y_2\colon k_2(y_2) = \{y_2\}\\ & \Rightarrow \forall x\colon P(k_1(x)) \Leftrightarrow P(k_2(x)).\end{aligned}$$

which is now in a form where it is possible to apply the instances of extensionality to the equalities in the newly introduced definitions. In order to preserve satisfiability of the proof obligation, we have to add as hypotheses instances of extensionality contraction for every pair of definitions where extensionality

expansion was applied. The final equi-satisfiable formula in basic normal form is

$$\begin{aligned}
&\wedge\ \forall z, y\colon z \in k_1(y) \Leftrightarrow z = y \vee z = y \\
&\wedge\ \forall z, y\colon z \in k_2(y) \Leftrightarrow z = y \\
&\wedge\ \forall y_1, y_2\colon (\forall z\colon z \in k_1(y_1) \Leftrightarrow z \in k_2(y_2)) \Rightarrow k_1(y_1) = k_2(y_2) \\
&\Rightarrow \forall x\colon P(k_1(x)) \Leftrightarrow P(k_2(x)).
\end{aligned}$$

3.3.4 Eliminating definitions. To improve the encoding, we introduce a procedure that eliminates definitions, having the opposite effect of the abstraction method where definitions are introduced and afterwards expanded to basic expressions. This process collects definitions of the form $x = \psi$, and then simply substitutes every occurrence of the term x by the non-basic expression ψ in the rest of the context, by applying the equality oriented as the rewriting rule $x \longrightarrow \psi$. The definitions we want to eliminate typically occur in the original proof obligation, meaning that they are not artificially introduced. In the following subsection, we will explain the interplay between normalization, definition abstraction, and definition elimination.

This transformation produces expressions that can eventually be normalized to their basic form. The restriction that x does not occur in ψ avoids rewriting loops and ensures termination of this process. For instance, the two equations $x = y$ and $y = x + 1$ will be transformed into $y = y + 1$, which cannot further be rewritten.[5] After applying the substitution, we can safely discard from the resulting formula the definition $x = \psi$, when x is a variable. However, we must keep the definition if x is an applied operator. Suppose we discard an assumption DOMAIN $f = S$, where the conclusion is $f \in [S \rightarrow T]$. Only after applying the rewriting rules, the conclusion will be expanded to an expression containing DOMAIN f, but the discarded fact required to simplify it to S will be missing.

3.3.5 Preprocessing algorithm. Now we can put together the encoding techniques described above in a single algorithm that we call *Preprocess*.

$$\begin{aligned}
\mathit{Preprocess}(\phi) &\triangleq \phi \\
&\quad \triangleright\ \mathit{Boolify} \\
&\quad \triangleright\ \textsc{Fix}\ \mathit{Reduce}
\end{aligned}
\qquad
\begin{aligned}
\mathit{Reduce}(\phi) &\triangleq \phi \\
&\quad \triangleright\ \textsc{Fix}\ (\mathit{Eliminate} \circ \mathit{Rewrite}) \\
&\quad \triangleright\ \textsc{Fix}\ (\mathit{Abstract} \circ \mathit{Rewrite})
\end{aligned}$$

Here, FIX $\mathcal{A}$ means that step $\mathcal{A}$ is executed until reaching a fixed point, the combinator $\triangleright$, used to chain actions on a formula ϕ, is defined as $\phi \triangleright f \triangleq f(\phi)$, and function composition $\circ$ is defined as $f \circ g \triangleq \lambda\phi.\, g(f(\phi))$.

The *Preprocess* algorithm takes a TLA$^+$ formula ϕ, Boolifies it and then applies repeatedly the step called *Reduce*, until reaching a fixed point, to transform

[5] The problem of efficiently eliminating definitions from propositional formulas is a major open question in the field of proof complexity. The definition-elimination procedure can result in an exponential increase in the size of the formula when applied naïvely [1].

the formula into a basic normal form. Only then the resulting formula is ready to be translated to the target language using the embedding of Section 3.2. In turn, *Reduce* first eliminates the definitions in the given formula (Sect. 3.3.4), applies the rewriting rules (Sect. 3.3.1) repeatedly, and then applies abstraction (Sect. 3.3.3) followed by rewriting repeatedly. Observe that the elimination step is in some sense opposite to the abstraction step: the first one eliminates every definition $x = \psi$ by using it as the rewriting rule $x \longrightarrow \psi$, while the latter introduces a new symbol x in the place of an expression ψ and asserts $x = \psi$, where ψ is non-basic in both cases. Therefore, elimination should only be applied before abstraction, and each of those should be followed by rewriting.

The *Preprocess* algorithm is sound because it is composed of sound substeps. It also terminates, meaning that it will always compute a basic normal formula, but with a caveat: we have to be careful that *Abstract* and *Eliminate* do not repeatedly act on the same expression. *Eliminate* does not produce non-basic expressions, but *Abstract* generates definitions that can be processed by *Eliminate*, reducing them again to the original non-basic expression. That is the reason for *Rewrite* to be applied after every application of *Abstract*: the new definitions are rewritten, usually by an extensionality expansion rule. In short, termination depends on the existence of extensionality rewriting rules for each kind of non-basic expression that *Abstract* may catch. Then, for any TLA^+ expression there exists an equi-satisfiable basic expression in normal form that the algorithm will compute.

3.4 Encoding CHOOSE

The CHOOSE operator is notoriously difficult for automatic provers to reason about. Nevertheless, we can exploit CHOOSE expressions by using the axioms that define them. By introducing a definition for $\text{CHOOSE } x\colon P(x)$, we obtain the theorem $\big(y = \text{CHOOSE } x\colon P(x)\big) \Rightarrow \big((\exists x\colon P(x)) \Leftrightarrow P(y)\big)$, where y is some fresh symbol. This theorem can be conveniently used as a rewriting rule after abstraction of CHOOSE expressions, and for CHOOSE expressions that occur negatively, in particular, as hypotheses of proof obligations.

For determinism of choice (axiom (2)), suppose an arbitrary pair of CHOOSE expressions $\phi_1 \triangleq \text{CHOOSE } x\colon P(x)$ and $\phi_2 \triangleq \text{CHOOSE } x\colon Q(x)$ where the free variables of ϕ_1 are $x_1, \ldots, x_n$ (noted $\mathbf{x}$) and those of ϕ_2 are $y_1, \ldots, y_m$ (noted $\mathbf{y}$). We need to check whether formulas P and Q are equivalent for every pair of expressions ϕ_1 and ϕ_2 occurring in a proof obligation. By abstraction of ϕ_1 and ϕ_2, we obtain the axiomatic definitions $\forall \mathbf{x}\colon f_1(\mathbf{x}) = \text{CHOOSE } x\colon P(x)$ and $\forall \mathbf{y}\colon f_2(\mathbf{y}) = \text{CHOOSE } x\colon Q(x)$, where f_1 and f_2 are fresh operator symbols of suitable arity. Then, we state the extensionality property for the pair f_1 and f_2 as the axiom $\forall \mathbf{x}, \mathbf{y}\colon \big(\forall x\colon P(x) \Leftrightarrow Q(x)\big) \Rightarrow f_1(\mathbf{x}) = f_2(\mathbf{y})$.

4 Evaluation

In order to validate our approach we reproved several test cases that had been proved interactively using the previously available TLAPS back-end provers, i.e.,

		ZIP	CVC4		Z3	
	size		u	t	u	t
Peterson	3	-	0.41	0.46	0.34	0.40
Peterson	10	5.69	0.78	0.96	0.80	0.97
Bakery	16	-	-	6.57	-	7.15
Bakery	223	52.74				
Memoir-T	1	-	-	-	1.99	1.53
Memoir-T	12	-	3.11	3.46	3.21	3.51
Memoir-T	424	7.31				
Memoir-I	8	-	3.84	5.79	9.35	10.23
Memoir-I	61	8.20				
Memoir-A	27	-	11.31	14.36	11.46	14.30
Memoir-A	126	19.10				

Finite sets	ZIP		Zenon+SMT		
	size		size	u	t
CardZero	11	5.42	5	0.48	0.48
CardPlusOne	39	5.35	3	0.49	0.52
CardOne	6	5.36	1	0.35	0.35
CardOneConv	9	0.63	2	0.35	0.36
FiniteSubset	62	7.16	19	-	5.77
PigeonHole	42	7.07	20	7.01	7.22
CardMinusOne	11	5.44	5	0.75	0.73

Table 1. Evaluation benchmarks results. An entry with the symbol "-" means that the solver has reached the timeout without finding the proof for at least one of the proof obligations. The backends were executed with a timeout of 300 seconds.

Zenon, Isabelle/TLA$^+$ and the decision procedure for Presburger arithmetic. We will refer to the combination of those three backends as ZIP for short.

For each benchmark, we compare two dimensions of an interactive proof: size and time. We define the *size* of an interactive proof as the number of non-trivial proof obligations generated by the Proof Manager. This number is proportional to the number of interactive steps and therefore represents the user effort for making TLAPS check the proof. The *time* is the number of seconds required by the Proof Manager to verify those proofs on a standard laptop.

Table 1 presents the results for four case studies: mutual exclusion proofs of the Peterson and Bakery algorithms, type-correctness and refinement proofs of the Memoir security architecture [8], and proofs of theorems about the cardinality of finite sets. We compare how proofs of different sizes are handled by the backends. Each line corresponds to an interactive proof of a given size. Columns correspond to the running times for a given SMT solver, where each prover is executed on all generated proof obligations. For our tests we have used the state-of-the-art SMT solvers CVC4 v1.3 and Z3 v4.3.0. For each prover we present two different times corresponding to the untyped encoding (the column labeled `u`) and the optimized encoding using the type system with refinement types [16] (labeled `t`).

In all cases, the use of the new backend leads to significant reductions in proof sizes and running times compared to the original interactive proofs. In particular, the "shallow" proofs of the first three case studies required only minimal interaction. We have also used the new SMT backend with good success on several proofs not shown here. Both SMT solvers offer similar results, with Z3 being better at reasoning about arithmetic. In a few cases CVC4 is faster or even proves obligations on which Z3 fails. Some proof obligations can be proved only by Zenon, in the case of big structural high-level formulas, or only using the "typed" encoding, because heavy arithmetic reasoning is required.

5 Conclusions

We have presented a sound and effective way of discharging TLA$^+$ proof obligations using automated theorem provers based on many-sorted first-order logic. This encoding was implemented in a back-end prover that integrates external SMT solvers as oracles to the TLA$^+$ Proof System TLAPS. The main component of the backend is a generic translation framework that makes available to TLAPS any SMT solver that supports the de facto standard format SMT-LIB/AUFLIA. We have also used the same framework for integrating automated theorem provers based on unsorted FOL, such as those based on the superposition calculus.

The resulting translation can handle a useful fragment of the TLA$^+$ language, including set theory, functions, linear arithmetic expressions, and the CHOOSE operator (Hilbert's choice). Encouraging results show that SMT solvers significantly reduce the effort of interactive reasoning for verifying "shallow" TLA$^+$ proof obligations, as well as some more involved formulas including linear arithmetic expressions. Both the size of the interactive proof, which reflects the number of user interactions, and the time required to find automatic proofs can be remarkably reduced with the new back-end prover.

The mechanism that combines term-rewriting with abstraction enables the backend to successfully handle CHOOSE expressions, tuples, records, and TLA$^+$ functions (λ-abstractions with domains). However, our rewriting method may introduce many additional quantifiers, which can be difficult for the automated provers to handle.

The untyped universe of TLA$^+$ is represented as a universal sort in MS-FOL. Purely set-theoretic expressions are mapped to formulas over uninterpreted symbols, together with relevant background axioms. The built-in integer sort and arithmetic operators are homomorphically embedded into the universal sort, and type inference is in essence delegated to the solver. The soundness of the encoding is immediate: all the axioms about sets, functions, records, tuples, etc. are theorems in the background theory of TLA$^+$ that exist in the Isabelle encoding. The "lifting" axioms for the encoding of arithmetic assert that TLA$^+$ arithmetic coincides with SMT arithmetic over integers. For ensuring completeness of our encoding, we would have to include the standard axiom of set extensionality in the background theory. For efficiency reasons, we include only instances of extensionality for specific sets, function domains, and functions.

The translation presented here forms the basis for further optimizations. In [16] we have explored the use of (incomplete) type synthesis for TLA$^+$ expressions, based on a type system with dependent and refinement types. Extensions for reasoning about real arithmetic and finite sequences would be useful. More importantly, we rely on the soundness of external provers, temporarily including them as part of TLAPS's trusted base. In future work we intend to reconstruct within Isabelle/TLA$^+$ (along the lines presented in [4]) the proof objects that many SMT solvers can produce. Such a reconstruction would have to take into account not only the proofs generated by the solvers, but also all the steps performed during the translation, including rewriting and abstraction.

Acknowledgements. We thank anonymous reviewers for their helpful comments.

References

1. J. Avigad. Eliminating definitions and Skolem functions in first-order logic. *ACM Trans. Comput. Logic*, 4(3):402–415, July 2003.
2. F. Baader and T. Nipkow. *Term rewriting and all that.* Cambridge University Press, 1999.
3. C. Barrett, A. Stump, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010.
4. J. C. Blanchette, S. Böhme, and L. C. Paulson. Extending Sledgehammer with SMT solvers. *Journal of Automated Reasoning*, 51(1):109–128, 2013.
5. R. Bonichon, D. Delahaye, and D. Doligez. Zenon: An extensible automated theorem prover producing checkable proofs. In N. Dershowitz and A. Voronkov, editors, *14th Intl. Conf. LPAR*, volume 4790 of *LNCS*, pages 151–165. Springer, 2007.
6. D. Déharbe, P. Fontaine, Y. Guyot, and L. Voisin. SMT solvers for Rodin. In *3rd Intl. Conf. Abstract State Machines, Alloy, B, VDM, and Z (ABZ 2012)*, volume 7316 of *LNCS*, pages 194–207, Pisa, Italy, 2012. Springer.
7. D. Delahaye, D. Doligez, F. Gilbert, P. Halmagrand, and O. Hermant. Zenon Modulo: When Achilles outruns the tortoise using deduction modulo. In K. McMillan, A. Middeldorp, and A. Voronkov, editors, *LPAR*, volume 8312 of *LNCS/ARCoSS*, pages 274–290, South Africa, 2013. Springer.
8. J. R. Douceur, J. R. Lorch, B. Parno, J. Mickens, and J. M. McCune. Memoir—Formal Specs and Correctness Proofs. Technical Report MSR-TR-2011-19, Microsoft Research, 2011.
9. M. Jacquel, K. Berkani, D. Delahaye, and C. Dubois. Tableaux modulo theories using superdeduction: An application to the verification of B proof rules with the Zenon automated theorem prover. In *Proc. of the 6th Intl. Joint Conf. on Automated Reasoning*, IJCAR'12, pages 332–338, Berlin, 2012. Springer.
10. M. Konrad and L. Voisin. Translation from set-theory to predicate calculus. Technical report, ETH Zurich, 2012.
11. L. Lamport. *Specifying Systems: The* TLA$^+$ *Language and Tools for Hardware and Software Engineers.* Addison-Wesley, Boston, Mass., 2002.
12. M. Manzano. *Extensions of First-Order Logic.* Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2nd edition, 2005.
13. D. Mentré, C. Marché, J.-C. Filliâtre, and M. Asuka. Discharging proof obligations from Atelier B using multiple automated provers. In *Proc. of the 3rd Intl. Conf. ABZ*, volume 7316 of *LNCS*, pages 238–251. Springer, 2012.
14. S. Merz and H. Vanzetto. Automatic verification of TLA$^+$ proof obligations with SMT solvers. In N. Bjorner and A. Voronkov, editors, *LPAR*, volume 7180 of *Lecture Notes in Computer Science*, pages 289–303. Springer, 2012.
15. S. Merz and H. Vanzetto. Harnessing SMT Solvers for TLA$^+$ Proofs. *ECEASST*, 53, 2012.
16. S. Merz and H. Vanzetto. Refinement Types for TLA$^+$. In J. Badger and K. Rozier, editors, *NASA Formal Methods*, volume 8430 of *Lecture Notes in Computer Science*, pages 143–157. Springer International Publishing, 2014.
17. J. Urban. Translating Mizar for first-order theorem provers. In A. Asperti, B. Buchberger, and J. Davenport, editors, *Mathematical Knowledge Management*, volume 2594 of *LNCS*, pages 203–215. Springer, 2003.