HAL
archives-ouvertes.fr

Encoding TLA+ into unsorted and many-sorted first-order logic

Stephan Merz, Hernán Vanzetto

▶ **To cite this version:**

Stephan Merz, Hernán Vanzetto. Encoding TLA+ into unsorted and many-sorted first-order logic. Science of Computer Programming, Elsevier, 2018, 158, pp.3-20. <10.1016/j.scico.2017.09.004>. <hal-01768750>

HAL Id: hal-01768750
https://hal.inria.fr/hal-01768750

Submitted on 17 Apr 2018

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$^+$ into Unsorted and Many-Sorted First-Order Logic[☆]

Stephan Merz[a], Hernán Vanzetto[b]

[a] *University of Lorraine, CNRS, Inria, LORIA, Nancy, France*
[b] *Yale University, New Haven, CT, United States*

Abstract

TLA$^+$ is a specification language designed for the verification of concurrent and distributed algorithms and systems. We present an encoding of a non-temporal fragment of TLA$^+$ into (unsorted) first-order logic and many-sorted first-order logic, the input languages of first-order automated theorem provers. The non-temporal subset of TLA$^+$ is based on untyped set theory and includes functions, arithmetic expressions, and Hilbert's choice operator. The translation, based on encoding techniques such as boolification, injection of unsorted expressions into sorted languages, term rewriting, and abstraction, is the core component of a back-end prover based on first-order theorem provers and SMT solvers for the TLA$^+$ Proof System.

Keywords: interactive theorem proving, set theory, many-sorted first-order logic, TLA$^+$, formal verification of distributed systems.

1. Introduction

TLA$^+$ [22] is a specification language designed for the verification of concurrent and distributed algorithms and systems. Its logical foundation is a combination of variants of Zermelo-Fraenkel set theory with choice (ZFC) and of linear-time temporal logic for modeling, respectively, the data manipulated by an algorithm, and its behavior. The TLA$^+$ Proof System (TLAPS) is an interactive proof assistant that provides support for mechanized reasoning about TLA$^+$ specifications. It integrates several back-end provers for making automatic reasoners available to users of TLAPS. More specifically, TLAPS is built around a so-called Proof Manager [11] that interprets a TLA$^+$ proof, generates corresponding proof obligations, and passes them to the back-end provers, which interact with external automated verifiers. The work reported here is motivated by the development of powerful back-end provers through which users of TLAPS interact with off-the-shelf automated theorem provers for non-temporal

[☆]This work was partially supported by the MSR-Inria Joint Centre and by project ANR-13-IS02-0001 of the Agence Nationale de la Recherche.

TLA$^+$ reasoning. In particular, we focus on two kinds of automated theorem provers: SMT (satisfiability modulo theories) solvers, and provers based purely on first-order logic (FOL), which we call FOL provers, such as those based on the superposition calculus.

Prior to this work, three back-end provers with different capabilities were available for non-temporal reasoning: 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, a tableau prover for first-order logic with equality that includes extensions for reasoning about sets and functions; and a decision procedure for Presburger arithmetic called SimpleArithmetic (now deprecated). The Isabelle and Zenon backends have very limited support for arithmetic reasoning, while SimpleArithmetic handled 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 FOL provers and SMT solvers for non-temporal proof obligations arising in TLAPS.[1] When verifying distributed algorithms, proof obligations are often "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. Sets and functions are at the core of modeling data in the TLA$^+$ language. Tuples and records, which occur very often in TLA$^+$ specifications, are also functions. Proof obligations corresponding to non-temporal steps of TLA$^+$ proofs are typically assertions that mix first-order logic with sets, functions, and arithmetic expressions. Accordingly, we do not aim at proofs of deep theorems of mathematical set theory but at good automation for formulas involving elementary set expressions, functions, records, and integer arithmetic.

The de-facto standard input languages are TPTP-FOF [34] for FOL provers and SMT-LIB [7], based on many-sorted FOL (MS-FOL [24]), for SMT solvers.[2] Initially, our goal was to target only SMT solvers, but with minor modifications we could adapt our translation to output untyped FOL. This translation, presented in Section 3, forms the core of the FOL and of the SMT backends. Although some of our encoding techniques 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;

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

[2] In this paper we use the terms *type* and *sort* interchangeably.

they denote some (unspecified) value. TLA$^+$ does not even distinguish between Boolean and non-Boolean expressions as in standard first-order logic, hence Boolean values can be stored in data structures just like any other value.

- 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 arguments that denote numbers.

- TLA$^+$ is equipped with a deterministic choice operator (Hilbert's epsilon operator), which has to be soundly encoded.

The first item is particularly challenging for our objectives: whereas an untyped language like TLA$^+$ is very expressive and flexible for writing specifications, automated reasoners rely on types for good automation, either internally [37] or explicitly, as in the case of SMT solvers. In order to support TLA$^+$ expressions in standard logics with terms, predicates and functions, we introduce a "boolification" step for distinguishing between Boolean and non-Boolean expressions. In a many-sorted environment, we use a single sort for encoding non-Boolean TLA$^+$ expressions, but we will introduce axioms that are used for reflecting expressions into interpreted sorts (such as integer arithmetic) that are natively supported by automatic reasoners. We therefore call this translation the "untyped" encoding of TLA$^+$; it essentially delegates type inference of sorted expressions to the external provers. Although there exist languages that are based on typed versions of set theory [1, 30], justifications for untyped specification languages are given by Lamport and Paulson [23]. TLAPS supports the existing TLA$^+$ language, and changing its semantic foundations was not an option in our work. We also believe that some of the techniques that we develop can be interesting for developing automated reasoning support for other systems based on standard, untyped set theory [16, 32]. In a separate paper [29], we discuss the orthogonal aspect of inferring types, which can be used to optimize the encoding given to the automated reasoners.

This article extends our paper published at the ABZ 2016 conference [28] by including the translation to unsorted FOL, additional examples, more details on proofs, algorithms, and encoding of operators, and in the evaluation of the new backends in the Paxos algorithm. It also supersedes our previous publications [26, 27], where we presented a primitive encoding of TLA$^+$ into SMT-LIB that did not explicitly address boolification, normalization and abstraction, and did not fully support CHOOSE expressions.

The article is structured as follows: Section 2 describes the underlying logic of TLA$^+$, Section 3 is the core of the paper and explains the encoding, Section 4 provides experimental results, Section 5 discusses related work, and Section 6 concludes and gives directions for future work.

2. A non-temporal fragment of TLA$^+$

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 deterministic choice operator. For a complete presentation of the TLA$^+$ language see [22, Sec. 16].

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

$$
\begin{array}{lll}
e ::= & v \mid w(e, \ldots, e) & \text{(terms)} \\
& \mid \text{FALSE} \mid e \Rightarrow e \mid \forall v\colon e \mid e = e \mid e \in e & \text{(formulas)} \\
& \mid \{\} \mid \{e, \ldots, e\} \mid \text{SUBSET } e \mid \text{UNION } e & \\
& \mid \{v \in e : e\} \mid \{e : v \in e\} & \text{(sets)} \\
& \mid \text{CHOOSE } x\colon e & \text{(choice)} \\
& \mid e[e] \mid \text{DOMAIN } e \mid [v \in e \mapsto e] \mid [e \to e] & \text{(functions)} \\
& \mid 0 \mid 1 \mid 2 \mid \ldots \mid \mathit{Int} \mid -e \mid e + e \mid e < e \mid e \mathinner{..} e & \text{(arithmetic)} \\
& \mid \text{IF } e \text{ THEN } e \text{ ELSE } e & \text{(conditional)}
\end{array}
$$

A *term* is a variable symbol v in $\mathcal{V}$ or an application of an operator symbol w in $\mathcal{O}$ to expressions, consistent with the arity of w. *Formulas* are built from FALSE, implication and universal quantification, and from the binary operators $=$ and $\in$. From these formulas, we can define the constant TRUE, the unary $\neg$, the binary connectives $\wedge$, $\vee$, $\Leftrightarrow$, and the existential quantifier $\exists$. Also, $\forall x \in S\colon e$ is defined as $\forall x\colon x \in S \Rightarrow e$.

TLA$^+$ has explicit syntax for *set objects* (empty set, enumeration, power set, generalized union, and two forms of set comprehension derived from the standard axiom schema of replacement) that are governed by the following axioms:

$$
\begin{array}{lll}
\text{(extensionality)} & (\forall x\colon x \in S \Leftrightarrow x \in T) \Rightarrow S = T & (2.1) \\
\text{(empty set)} & x \in \{\} \Leftrightarrow \text{FALSE} & (2.2) \\
\text{(enumeration)} & x \in \{e_1, \ldots, e_n\} \Leftrightarrow x = e_1 \vee \ldots \vee x = e_n & (2.3) \\
\text{(power set)} & S \in \text{SUBSET } T \Leftrightarrow \forall x \in S\colon x \in T & (2.4) \\
\text{(union)} & x \in \text{UNION } S \Leftrightarrow \exists T \in S\colon x \in T & (2.5) \\
\text{(comprehension}_1\text{)} & x \in \{y \in S\colon P(y)\} \Leftrightarrow x \in S \wedge P(x) & (2.6) \\
\text{(comprehension}_2\text{)} & x \in \{e(y)\colon y \in S\} \Leftrightarrow \exists y \in S\colon x = e(y) & (2.7)
\end{array}
$$

[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.

We implicitly consider the universal closures of the above axioms, except for P and e in the comprehension axioms that are schematic variables, meaning that they can be instantiated by countably infinite expressions.[4]

Another primitive construct of TLA$^+$ is Hilbert's choice operator, written CHOOSE $x : P(x)$, 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 $x : P(x)$ is some fixed, but unspecified value. CHOOSE satisfies the following axiom schemas. The first one gives an alternative way of defining quantifiers, and the second one expresses that CHOOSE is deterministic.

$$\big(\exists x : P(x)\big) \;\Leftrightarrow\; P\big(\text{CHOOSE } x : P(x)\big) \tag{2.8}$$

$$\big(\forall x : P(x) \Leftrightarrow Q(x)\big) \;\Rightarrow\; \big(\text{CHOOSE } x : P(x)\big) = \big(\text{CHOOSE } x : Q(x)\big) \tag{2.9}$$

From axiom (2.9) note that if there is no value satisfying some predicate P, then (CHOOSE $x : P(x)$) = (CHOOSE $x :$ FALSE). Consequently, the expression CHOOSE $x :$ FALSE and all its equivalent forms represent a unique value.

Certain TLA$^+$ values are *functions*. Unlike standard ZFC set theory, TLA$^+$ functions are not identified with 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

$$f = [x \in S \mapsto e] \;\Leftrightarrow\; IsAFcn(f) \wedge \text{DOMAIN } f = S \wedge \forall x \in S : f[x] = e \tag{2.10}$$

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

As a set theoretic language, every TLA$^+$ expression—including formulas, functions, and numbers—denotes a set.

3. Untyped encoding into FOL and MS-FOL

We define a translation from TLA$^+$ to our target languages as follows: given a TLA$^+$ proof obligation, we generate an equi-satisfiable collection of either TPTP [34] formulas (in the first-order form, or FOF, dialect) or SMT-LIB [7]

[4]Both axioms (2.6) and (2.7) for set comprehension objects are instances of the standard axiom schema of replacement: taking the two single-valued predicates $\phi_1(y, x) \triangleq x = y \wedge P(y)$ and $\phi_2(y, x) \triangleq x = e(y)$, we can define $\{y \in S : P(y)\} \triangleq \mathcal{R}(S, \phi_1)$ and $\{e(y) : y \in S\} \triangleq \mathcal{R}(S, \phi_2)$. The replacement axiom says that, given an expression S and a binary predicate ϕ, such that ϕ is *single-valued* for any y in S, that is, $\forall y \in S : \forall x, z : \phi(y, x) \wedge \phi(y, z) \Rightarrow x = z$, then there exists a set object $\mathcal{R}(S, \phi)$, and that $x \in \mathcal{R}(S, \phi) \Leftrightarrow \exists y \in S : \phi(y, x)$ [32].

formulas (in the AUFLIA logic) whose proof can be attempted respectively by FOL provers or SMT solvers.

First, we identify which expressions are used as propositions and translate them to formulas. All non-Boolean expressions, including sets, functions, and numbers, are represented by terms (Section 3.1). In the case of SMT-LIB, we declare a new sort U, for TLA$^+$ universe, as the target of translating these expressions.

We then proceed in two main steps. Satisfiability-preserving transformations are applied during a pre-processing phase in order to remove expressions not supported by the target languages (Section 3.2). The result is an intermediate *basic* TLA$^+$ formula, *i.e.*, a TLA$^+$ expression that has an obvious counterpart in TPTP or SMT-LIB. We define basic TLA$^+$ as a subset of TLA$^+$ consisting of terms, formulas, equality and set membership relations, and IF-THEN-ELSE expressions. When the target is MS-FOL, basic TLA$^+$ aditionally includes primitive arithmetic operators. This step is the exactly same for both target languages. The second step is a shallow embedding of basic expressions into either FOL or MS-FOL (Section 3.3).

Finally, we explain how the encoding of functions (Section 3.4), tuples and records (Section 3.5), CHOOSE expressions (Section 3.6), strings and IF-THEN-ELSE expressions (Section 3.7) fit in the preprocessing phase of the translation.

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. TLAPS adopts the so-called liberal interpretation of TLA$^+$ Boolean expressions [22, Sec. 16.1.3] where any expression whose top-level connective is a logical operator, $=$ or $\in$ has a Boolean value.[5] Moreover, the interpretation of any expression with a top-level logical connective agrees with the interpretation of the expression obtained by replacing every argument e of that connective with the expression $e = \text{TRUE}$.

For example, consider $\forall x \colon (\neg\neg x) = x$, 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, $\forall x \colon (\neg\neg x) \Leftrightarrow x$ is valid because it is interpreted as $\forall x \colon (\neg\neg(x = \text{TRUE})) \Leftrightarrow (x = \text{TRUE})$. Observe that the value of $x = \text{TRUE}$ is a Boolean for any x, although the truth value is unspecified if x is non-Boolean.

In order to identify the expressions used as propositions we use the simple algorithm of Figure 1, which is mutually defined by the operator $[\![e]\!]^+$ that is

[5] The semantics of TLA$^+$ offers three alternatives to interpret expressions [22, 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. Only in the moderate and liberal interpretation, the expression 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.

$$
\begin{array}{rcl@{\qquad}rcl}
[\![x]\!]^+ & \triangleq & x^b & [\![w(\vec{e})]\!]^+ & \triangleq & w^b([\![\vec{e}]\!]^-) \\
[\![e_1 \Rightarrow e_2]\!]^+ & \triangleq & [\![e_1]\!]^+ \Rightarrow [\![e_2]\!]^+ & [\![\forall x\colon e]\!]^+ & \triangleq & \forall x\colon [\![e]\!]^+ \\
[\![e_1 = e_2]\!]^+ & \triangleq & [\![e_1]\!]^- = [\![e_2]\!]^- & [\![e_1 \in e_2]\!]^+ & \triangleq & [\![e_1]\!]^- \in [\![e_2]\!]^- \\
[\![e_1[e_2]]\!]^+ & \triangleq & ([\![e_1]\!]^-[[\![e_2]\!]^-])^b & [\![\mathcal{E}]\!]^+ & \triangleq & \mathit{error} \\
\multicolumn{6}{c}{[\![\text{IF } e_1 \text{ THEN } e_2 \text{ ELSE } e_3]\!]^+ \triangleq \text{IF } [\![e_1]\!]^+ \text{ THEN } [\![e_2]\!]^+ \text{ ELSE } [\![e_3]\!]^+} \\
\multicolumn{6}{c}{[\![\text{CHOOSE } x\colon e]\!]^+ \triangleq (\text{CHOOSE } x\colon [\![e]\!]^+)^b}
\end{array}
$$

$$
\begin{array}{rcl@{\qquad}rcl}
[\![x]\!]^- & \triangleq & x & [\![w(\vec{e})]\!]^- & \triangleq & w([\![\vec{e}]\!]^-) \\
[\![e_1 \Rightarrow e_2]\!]^- & \triangleq & [\![e_1 \Rightarrow e_2]\!]^+ & [\![\forall x\colon e]\!]^- & \triangleq & [\![\forall x\colon e]\!]^+ \\
[\![e_1 = e_2]\!]^- & \triangleq & [\![e_1 = e_2]\!]^+ & [\![e_1 \in e_2]\!]^- & \triangleq & [\![e_1 \in e_2]\!]^+ \\
[\![e_1[e_2]]\!]^- & \triangleq & [\![e_1]\!]^-[[\![e_2]\!]^-] & [\![\text{DOMAIN } e]\!]^- & \triangleq & \text{DOMAIN } [\![e]\!]^- \\
[\![[x \in e_1 \mapsto e_2]]\!]^- & \triangleq & [x \in [\![e_1]\!]^- \mapsto [\![e_2]\!]^-] & [\![\{\vec{e}\}]\!]^- & \triangleq & \{[\![\vec{e}]\!]^-\} \\
[\![\{x \in e_1 : e_2\}]\!]^- & \triangleq & \{x \in [\![e_1]\!]^- : [\![e_2]\!]^+\} & [\![\text{UNION } e]\!]^- & \triangleq & \text{UNION } [\![e]\!]^- \\
[\![\{e_1 : x \in e_2\}]\!]^- & \triangleq & \{[\![e_1]\!]^- : x \in [\![e_2]\!]^-\} & [\![\text{SUBSET } e]\!]^- & \triangleq & \text{SUBSET } [\![e]\!]^- \\
\multicolumn{6}{c}{[\![\text{IF } e_1 \text{ THEN } e_2 \text{ ELSE } e_3]\!]^- \triangleq \text{IF } [\![e_1]\!]^+ \text{ THEN } [\![e_2]\!]^- \text{ ELSE } [\![e_3]\!]^-} \\
\multicolumn{6}{c}{[\![\text{CHOOSE } x\colon e]\!]^- \triangleq \text{CHOOSE } x\colon [\![e]\!]^+}
\end{array}
$$

Figure 1: Boolification algorithm: $[\![e]\!]^+$ processes the expression e as a formula, attaching a ^{b} symbol when finding a term, a function application, or a CHOOSE, and $[\![e]\!]^-$ considers e as a non-Boolean expression (arithmetic expressions are omitted). Notation: v denotes a variable identifier, w is a polyadic operator, e, e_1, e_2 and e_3 are expressions, and the symbol $\mathcal{E}$ encompasses all non-Boolean expressions, such as set, function and arithmetic expressions.

applied when the expression e is considered as a formula, and by the operator $[\![e]\!]^-$ that is applied when e is considered a non-Boolean expression. The algorithm recursively traverses an expression searching for sub-expressions that should be treated as formulas. Sub-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 $\mathsf{boolify} : \mathsf{U} \to \mathsf{Bool}$. Analogously in TPTP we use a unary predicate. The above examples are translated as $\forall x^{\mathsf{U}}\colon (\neg\neg\mathsf{boolify}(x)) = x$ and $\forall x^{\mathsf{Bool}}\colon (\neg\neg x) \Leftrightarrow x$, revealing their (in)validity.

3.2. Preprocessing

Through a series of transformations applied to a boolified TLA$^+$ proof obligation, we obtain an equi-satisfiable formula that can be straightforwardly

passed to the external provers using the direct encoding of basic expressions described later (Section 3.3). The main motivation is to eliminate those TLA$^+$ expressions that cannot be expressed 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.

3.2.1. Normalization by rewriting.

We define a rewriting process that systematically expands definitions of non-basic constructs. Instead of letting the solver find instances of the background axioms introduced in Section 2, it applies the "obvious" instances of those axioms during the translation. For instance, the axioms (2.5) for the UNION operator and (2.6) for the first form of comprehension yield, respectively, the rewriting rules

$$\begin{aligned} x \in \text{UNION } S &\longrightarrow \exists T \in S : x \in T, \text{ and} \\ x \in \{y \in S : P(y)\} &\longrightarrow x \in S \wedge P(x). \end{aligned}$$

This simple process of normalization by rewriting is often enough for eliminating all non-basic constructs. The remaining cases are left to the abstraction mechanism described in the next subsection.

All rewriting rules defined in this paper apply equivalence-preserving transformations. To ensure soundness, we derive each rewriting rule from a theorem that we have proved in Isabelle/TLA$^+$. More specifically, the theorem corresponding to a rule $a \longrightarrow b$ is $\forall \mathbf{x} : a \Leftrightarrow b$ when a and b are Boolean expressions, and $\forall \mathbf{x} : a = b$ otherwise, where $\mathbf{x}$ denotes all free variables in the rule.

The standard ZF extensionality axiom for sets (2.1) 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 the extensionality property for equality expressions $x = y$ whenever x or y has a top-level operator that constructs a set. In these cases, we say that we *expand* equality. Specifically, for each set expression T we derive rewriting rules for equations $x = T$ and $T = x$. For instance, the rules

$$\begin{aligned} x = \{z_1, \ldots, z_n\} &\longrightarrow \forall z : z \in x \Leftrightarrow z = z_1 \vee \cdots \vee z = z_n \\ x = \text{UNION } S &\longrightarrow \forall z : z \in x \Leftrightarrow \exists T \in S : z \in T, \text{ and} \\ x = \{z \in S : P\} &\longrightarrow \forall z : z \in x \Leftrightarrow z \in S \wedge P \end{aligned}$$

are derived from set extensionality (2.1) and the axioms of set enumeration (2.3), UNION (2.5), and bounded set comprehension (2.6).

For example, in the formula

$$\{\} = \{z \in Nat : z < 0\}$$

the equality is rewritten by the expansion rule for set comprehension to

$$\forall z : z \in \{\} \Leftrightarrow z \in Nat \wedge z < 0.$$

By applying the rule $x \in \{\} \longrightarrow$ FALSE, we obtain the first-order formula

$$\forall z : \text{FALSE} \Leftrightarrow z \in Nat \wedge z < 0$$

whose proof relies on the techniques described in Section 3.3 for encoding arithmetic expressions.

Because extensionality is not applied in full generality, the translation becomes incomplete. Even assuming that the automated theorem provers are semantically complete, the translation of a semantically valid TLA^+ formula may become invalid when encoded. In these cases, the user will need to explicitly add the extensionality axiom as a hypothesis to the TLA^+ proof.

We also include a rule for the *contraction* of set extensionality:

$$(\forall z\colon z \in x \Leftrightarrow z \in y) \;\longrightarrow\; x = y \tag{3.1}$$

which we apply with higher priority than the expansion rules.

All rules of the form $a \longrightarrow b$, including those introduced below for functions and CHOOSE expressions, define a term rewriting system $(\text{TLA}^+, \longrightarrow)$, where $\longrightarrow$ is a binary relation over well-formed TLA^+ expressions. A comprehensive list of the rewriting rules can be found in Appendix A.

Before presenting proofs of two properties about $(\text{TLA}^+, \longrightarrow)$, we recall some standard concepts of rewriting systems (RS) [6]. A RS is a pair $(\text{A}, \longrightarrow)$ consisting of a set A of objects and a binary relation $\longrightarrow$ on A. The reflexive transitive closure of $\longrightarrow$ is noted $\xrightarrow{*}$, and we write $a \xrightarrow{*} b$ when there is some finite path from a to b. An RS is *terminating* if there is no infinite descending chain $a_0 \longrightarrow a_1 \longrightarrow \cdots$ of objects a_i. Then, the relation $\longrightarrow$ is *well-founded*.

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

Proof sketch. Termination is proved by embedding $(\text{TLA}^+, \longrightarrow)$ into a system that is known to have a well-founded ordering, typically $(\mathbb{N}, >)$ [6]. The embedding is through an ad-hoc monotone mapping μ such that $\mu(a) > \mu(b)$ for every rule $a \longrightarrow b$. We define it in such a way that every rule instance strictly decreases the number of non-basic and complex expressions such as quantifiers.

For that, we assign coefficients $\mathcal{W}(e)$ to every symbol and construct e through the relation $=_{\mathcal{W}}$, as shown in the Table 1. The lower the coefficient given to e, the more preference is given to e to be used in the right-hand side of the rules. Remember that what is considered non-basic, depends on the target language.

Let $e|_p$ denote the sub-expression of e at some position p of its syntactic tree. We define the embedding μ of an expression e as the sum of the coefficients $\mathcal{W}(e|_p)$ for every position p in e: $\mu(e) \triangleq \mathcal{W}(e|_1) + \ldots + \mathcal{W}(e|_n)$. Given this coefficient assignment, one can easily check in all rewriting rules that the weight of the left-hand side is strictly larger than that of the right-hand side. Therefore, it is possible to embed $(\text{TLA}^+, \longrightarrow)$ into a terminating system. □

Our next objective is to prove that $(\text{TLA}^+, \longrightarrow)$ is confluent. We say that two objects a_1 and a_2 are *joinable*, noted $a_1 \downarrow a_2$, if there is an object b such that $a_1 \xrightarrow{*} b$ and $a_2 \xrightarrow{*} b$. A RS is *confluent* if $a \xrightarrow{*} a_1$ and $a \xrightarrow{*} a_2$ implies $a_1 \downarrow a_2$, that is, when the system defines at most one normal form for each object. Newman's lemma [6] allows us to prove that a terminating RS is confluent by proving instead that the system is locally confluent, which is

$v, w(\_, \ldots, \_)$	$=_{\mathcal{W}}$ 0	$0, 1, 2, \ldots$	$=_{\mathcal{W}}$ 0
FALSE, TRUE	$=_{\mathcal{W}}$ 0	$+, -, *$	$=_{\mathcal{W}}$ 0
$\wedge, \vee, \Rightarrow, \Leftrightarrow, \neg, =$	$=_{\mathcal{W}}$ 0	$<, \leq, \geq, >$	$=_{\mathcal{W}}$ 0
$\neq$	$=_{\mathcal{W}}$ 1	Int, Nat	$=_{\mathcal{W}}$ 1
$\forall, \exists \_ : \_$	$=_{\mathcal{W}}$ 1	$\div, \%$	$=_{\mathcal{W}}$ 2
IF _ THEN _ ELSE _	$=_{\mathcal{W}}$ 1	$..$	$=_{\mathcal{W}}$ 2
$\in$	$=_{\mathcal{W}}$ 0	DOMAIN	$=_{\mathcal{W}}$ 1
$\notin$	$=_{\mathcal{W}}$ 1	$\_[\_]$	$=_{\mathcal{W}}$ 2
$\{\}, \{\_, \ldots, \_\}, \subseteq, \cup, \cap, \setminus$	$=_{\mathcal{W}}$ 2	$[\_ \in \_ \mapsto \_]$	$=_{\mathcal{W}}$ 2
SUBSET, UNION	$=_{\mathcal{W}}$ 3	$[\_$ EXCEPT $\_ = \_]$	$=_{\mathcal{W}}$ 3
$\{\_ \in \_ : \_\}, \{\_ : \_ \in \_\}$	$=_{\mathcal{W}}$ 3	$[\_ \to \_]$	$=_{\mathcal{W}}$ 3

Table 1: Weight coefficients of TLA^+ symbols and constructs for termination proof of $(\text{TLA}^+, \longrightarrow)$. Basic operators weigh 0, including $\in$ and arithmetic expressions, except for IF-THEN-ELSE expressions that are non-basic when translating to FOL. Non-basic expressions have a higher coefficient than quantifiers, some more than others, depending if they are expressed in terms of quantifiers or other non-basic expressions.

a weaker property. A relation $\longrightarrow$ is *locally confluent* if, for every object a, $a \longrightarrow a_1$ and $a \longrightarrow a_2$ implies $a_1 \downarrow a_2$, that is, it is confluent restricted to one-step divergences.

We still need another step to express the property of local confluence as one that can be broken down into simpler statements. The Critical Pair Theorem [18, 20] says that a term-rewriting system is locally confluent if and only if all its critical pairs are joinable. Consider any two rules $a_1 \longrightarrow b_1$ and $a_2 \longrightarrow b_2$ whose variables have been renamed such that the rules do not have variables in common. Let p be some non-root position in the syntactic tree of a_1 such that $a_1|_p$ is not a variable, and let σ be a most general unifier (mgu) of $a_1|_p$ and a_2, that is, the superposition of the left-hand sides of both rewriting rules. Then, the pair $\langle b_1\sigma; (a_1\sigma)[b_2\sigma]_p \rangle$ is called a *critical pair*, where $a[b]_p$ denotes the term a such that b replaces $a|_p$.[6] A critical pair may in particular arise by overlapping a rule with itself (with its variables renamed). The number of critical pairs is finite, and proving that they are joinable is decidable [6].

Theorem 2. $(\text{TLA}^+, \longrightarrow)$ *is confluent.*

Proof sketch. By Newman's lemma and the Critical Pair Theorem, and because $(\text{TLA}^+, \longrightarrow)$ is terminating, it suffices to prove that all critical pairs are joinable. By enumerating all combinations of rewriting rules, we can find all critical pairs $\langle e_1; e_2 \rangle$ between them. To prove that all e_1 and e_2 are joinable for each

[6] For example, in a term-rewriting system over standard FOL, consider the rules $f(f(x, y), z) \longrightarrow f(x, f(y, z))$ and $f(g(a), a) \longrightarrow b$, where f, g are functions and x, y, a are variables. By unifying the subterm $f(x, y)$ with $f(g(a), a)$, we obtain the mgu $\{x \mapsto g(a); y \mapsto a\}$ and their critical pair $\langle f(g(a), f(a, z)); f(b, z) \rangle$. An expression such as $f(f(g(a)), a), z)$ can be reduced to the two elements of the critical pair.

such pairs, we reduce them to their normal forms e_1' and e_2' and show that they are syntactically equal. In particular, the contraction rule (3.1) is necessary for obtaining a normalizing system.

We illustrate the procedure with one example. Consider the pair of overlapping rules $x \cup \{\} \longrightarrow x$ and

$$y = t \cup u \longrightarrow \forall z\colon z \in y \Leftrightarrow z \in t \vee z \in u \tag{3.2}$$

that yield the *mgu* $\{t \mapsto x; u \mapsto \{\}\}$ and the critical pair

$$\langle \forall z\colon z \in y \Leftrightarrow z \in x \vee z \in \{\}; y = x \rangle.$$

The first element of the pair is not yet in normal form, but $y = x$ is. By the rules $x \in \{\} \longrightarrow \text{FALSE}$, $\varphi \vee \text{FALSE} \longrightarrow \varphi$, and (3.1), it is possible to reach the same normal form as the second element. Therefore these two rules are joinable.

What this case shows is that there are two ways of reducing an expression such as $S = T \cup \{\}$ to its normal form, $S = T$. A possible one-step path is by the rule $x \cup \{\} \longrightarrow x$. The other choice is by first applying (3.2), yielding the path

$$\begin{array}{lll}
S = T \cup \{\} & \longrightarrow \forall z\colon z \in S \Leftrightarrow z \in T \vee z \in \{\} & \text{by (3.2)} \\
& \longrightarrow \forall z\colon z \in S \Leftrightarrow z \in T \vee \text{FALSE} & \text{by } x \in \{\} \longrightarrow \text{FALSE} \\
& \longrightarrow \forall z\colon z \in S \Leftrightarrow z \in T & \text{by } \varphi \vee \text{FALSE} \longrightarrow \varphi \\
& \longrightarrow S = T & \text{by (3.1)}
\end{array}$$

Similar reasoning can be applied to all critical pairs of $(\text{TLA}^+, \longrightarrow)$. □

3.2.2. 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 non-basic sub-expressions $\{x\} \cup \{x\}$ and $\{x\}$ do not occur in the form of a left-hand side of any rewriting rule, so 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 all occurrences of ψ with free variables $x_1, \ldots, x_n$, we introduce in their place a fresh term $k(x_1, \ldots, x_n)$, and add the formula $k(x_1, \ldots, x_n) = \psi$ 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 normalization. We ensure that non-basic expressions occurring more than once are replaced by the same 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

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

which is now in a form where it is possible to expand (using extensionality) 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

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

3.2.3. Eliminating definitions

Efficiently handling equality is very important in practice, and we rely on a procedure for eliminating definitions. This procedure has the opposite effect of the abstraction method where definitions are introduced and afterwards expanded to basic expressions. It collects all definitions of the form $x = \psi$, and then simply applies the rewriting rules $x \longrightarrow \psi$ to substitute every occurrence of the term x by the non-basic expression ψ in the rest of the context. The definitions we want to eliminate typically occur in the original proof obligation, that is, they do not result from the abstraction step.

This transformation produces expressions that can eventually be normalized to their basic form. To avoid rewriting loops and ensure termination, it can only be applied if x does not occur in ψ. For instance, the two equations $x = y$ and $y = x + 1$ will be transformed into $y = y + 1$, which cannot further be rewritten.[7] 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 a complex expression. Suppose we discard an assumption DOMAIN $f = S$, where the conclusion is $f \in [S \to T]$. 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.

3.2.4. Preprocessing algorithm

Now we can put together boolification and the encoding techniques described above in a single algorithm called `preprocess`.

$$\begin{array}{lll} \texttt{preprocess}(\phi) \triangleq & \phi & \\ & \triangleright\ \texttt{boolify} & \\ & \triangleright\ \textsc{Fix}\ \texttt{reduce} & \end{array} \qquad \begin{array}{lll} \texttt{reduce}(\phi) \triangleq & \phi & \\ & \triangleright\ \textsc{Fix}\ (\texttt{eliminate} \circ \texttt{rewrite}) & \\ & \triangleright\ \textsc{Fix}\ (\texttt{abstract} \circ \texttt{rewrite}) & \end{array}$$

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 $f \circ g \triangleq \lambda\phi.\, g(f(\phi))$.

[7] 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 [2].

Given a TLA$^+$ formula ϕ, the algorithm boolifies it and then applies repeatedly the step called `reduce` to obtain its basic normal form, which serves as the input for the embedding described in Section 3.3. In turn, `reduce` first eliminates the definitions in the given formula (Section 3.2.3) and applies the rewriting rules (Section 3.2.1) repeatedly, and then applies abstraction (Section 3.2.2) followed by rewriting repeatedly.

The `preprocess` algorithm is sound, because it is composed of sound substeps, and it terminates, meaning that it will always compute a basic normal formula.

Theorem 3. *The* `preprocess` *algorithm terminates.*

Proof idea. 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. That is why we apply elimination before abstraction, and why each of those is followed by rewriting. 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. □

3.3. Direct embedding

The preprocessing phase outputs a boolified basic TLA$^+$ expression that we will encode essentially using first-order formulas and uninterpreted functions, without substantially changing its structure. In short, the final step of the encoding maps the given basic expression to corresponding formulas in the target languages in an (almost) verbatim way. The direct embedding follows the same basic approach for single-sorted and many-sorted logics.

For first-order TLA$^+$ expressions it suffices to apply a shallow embedding into first-order formulas. When the target language is FOL, we will encode non-Boolean and Boolean operators respectively as functions and predicates, respecting their arities. For example, the primitive relation $\in$, which is the only set theoretic operator that can appear in a basic TLA$^+$ formula, will be represented by a predicate in of arity two. When the target language is sorted, non-logical TLA$^+$ operators will be declared as function or predicate symbols with U-sorted arguments. So for instance, $\in$ will be encoded in SMT-LIB as the function $\mathsf{in} : \mathsf{U} \times \mathsf{U} \to \mathsf{Bool}$.

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 operators and formulas that we have presented so far are expressed in FOL using uninterpreted function symbols over the sorts U and Bool.

We want to benefit from the native capabilities for arithmetic provided by the SMT solvers, but we cannot directly embbed the arithmetic expressions using the built-in Int type for integers of SMT solvers. For example, it would be unsound to represent the TLA$^+$ formula $x - 0 = x$ by assigning type Int to the variable x and using the built-in integer subtraction of the SMT solver to encode TLA$^+$'s subtraction operator. Instead, we declare an injective function $\mathsf{i2u} : \mathsf{Int} \to \mathsf{U}$ that embeds built-in integers into the sort U.[8] Integer literals k are simply encoded as $\mathsf{i2u}(k)$. For example, the formula $3 \in \mathit{Int}$ is translated as $\mathsf{in}(\mathsf{i2u}(3), \mathsf{tla\_Int})$, for which we have to declare $\mathsf{tla\_Int} : \mathsf{U}$ and add to the translation the axiom for Int:

$$\forall x^{\mathsf{U}} : \mathsf{in}(x, \mathsf{tla\_Int}) \Leftrightarrow \exists n^{\mathsf{Int}} : x = \mathsf{i2u}(n).$$

Observe that this axiom introduces two quantifiers in the results of our 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 existential quantifier remains. Arithmetic operators over TLA$^+$ values are defined homomorphically over the image of $\mathsf{i2u}$ by axioms such as

$$\forall m^{\mathsf{Int}}, n^{\mathsf{Int}} : \mathsf{plus}(\mathsf{i2u}(m), \mathsf{i2u}(n)) = \mathsf{i2u}(m + n), \tag{3.3}$$

where $+$ denotes the built-in addition over integers, and $\mathsf{plus} : \mathsf{U} \times \mathsf{U} \to \mathsf{U}$ represents the addition operator of TLA$^+$.

As a result, type inference in all these cases 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 $\mathsf{i2u}$.

If we call $\texttt{basic\_to\_fol}(\phi)$ and $\texttt{basic\_to\_msfol}(\phi)$ the embeddings of a basic TLA$^+$ formula ϕ into FOL and MS-FOL respectively, we can define the processes of encoding TLA$^+$ into FOL and MS-FOL as:

$$\begin{array}{lll} \texttt{tla\_to\_fol}(\phi) \triangleq & \phi & \\ & \triangleright\ \texttt{preprocess} & \\ & \triangleright\ \texttt{basic\_to\_fol} & \end{array} \qquad \begin{array}{lll} \texttt{tla\_to\_msfol}(\phi) \triangleq & \phi & \\ & \triangleright\ \texttt{preprocess} & \\ & \triangleright\ \texttt{basic\_to\_msfol} & \end{array}$$

In the following we provide two toy examples that illustrate these encodings.

3.3.1. Example: encoding into FOL

In TLA$^+$, the cardinality of finite sets is expressed using a unary constant operator called *Cardinality*. Given constant predicates *IsFiniteSet* and *IsBijection*

[8] 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 $\mathsf{u2i}$, we use instead the axiom $\forall n^{\mathsf{Int}} : \mathsf{u2i}(\mathsf{i2u}(n)) = n$, which generates a linear number of instances $\mathsf{i2u}(n)$, where $\mathsf{u2i} : \mathsf{U} \to \mathsf{Int}$ is unspecified.

```
1 axiom  isint(0)
2 axiom  isint(1)
3 axiom  ∀X, M, N. in(X, tla_Nat) ⇔ isint(X) ∧ leq(0, X)
4 axiom  ∀X, M, N. in(X, interval(M, N)) ⇔
                        isint(M) ∧ isint(N) ∧ isint(X)∧
                        leq(M, X) ∧ leq(X, M)
5 axiom  ∀S. boolify(isFiniteSet(S))
             ⇒ ∀N. N = cardinality(S)
                    ⇔ in(N, tla_Nat) ∧
                       ∃F. boolify(isBijection(F, interval(1, N), S))
6 conjecture  ∀S. boolify(isFiniteSet(S)) ⇒ in(cardinality(S), tla_Nat)
```

Figure 2: TPTP-FOF encoding of the obligation generated from the proof of *CardinalityInNat*.

(defined elsewhere), the semantics of *Cardinality* is defined with the following axiom:

$$\text{AXIOM } \mathit{CardinalityAxiom} \equiv \forall S : \mathit{IsFiniteSet}(S) \Rightarrow \forall n : (n = \mathit{Cardinality}(S)) \Leftrightarrow (n \in \mathit{Nat}) \wedge \exists f : \mathit{IsBijection}(f, 1..n, S)$$

Now consider the following lemma and its proof:

LEMMA $\mathit{CardinalityInNat} \triangleq \forall S : \mathit{IsFiniteSet}(S) \Rightarrow \mathit{Cardinality}(S) \in \mathit{Nat}$
BY $\mathit{CardinalityAxiom}, \mathit{FOL}$

The BY proof statement asserts that the lemma can be proved using *CardinalityAxiom* as a fact, and that the Proof Manager should attempt to find a proof by using the FOL back-end prover. The definitions for predicates *IsFiniteSet* and *IsBijection* are irrelevant for the proof. The Proof Manager will generate a proof obligation whose plain translation (without optimizations) to TPTP-FOF is presented in Figure 2. Lines 3 and 4 give the axioms for tla_Nat and interval, respectively. They are not required for the validity of the lemma, but we include them in the translation simply because the operators *Nat* and .. (for integer intervals) occur in the obligation. Note that the function leq representing the operator $\leq$ is left unspecified. Line 5 is the translation of *CardinalityAxiom*. Line 6 corresponds to the theorem's statement.

3.3.2. Example: encoding into MS-FOL

Consider the trivial TLA$^+$ proof obligation $\forall x \in \mathit{Int} : x + 0 = x$. Its translation to SMT-LIB is shown in Figure 3: line 4 states the injectivity of i2u, line 5 corresponds to the axiom of addition, and line 6 to the proper (negated) proof obligation. Let us illustrate the interplay of the axioms on this concrete example. By Skolemization on line 6, the solver introduces a new constant, say n, of

```
1 declare  i2u : (Int) U
2 declare  u2i : (U) Int
3 declare  plus : (U U) U
4 assert  ∀n^Int : u2i(i2u(n)) = n
5 assert  ∀m^Int, n^Int . plus(i2u(m), i2u(n)) = i2u(m + n)
6 assert ¬(∀x^U . (∃n^Int . x = i2u(n)) ⇒ plus(x, i2u(0)) = x)
```

Figure 3: SMT-LIB encoding of $\forall x \colon x \in \mathit{Int} \Rightarrow x + 0 = x$ (in a pretty-printed presentation).

sort Int, such that $x = \mathsf{i2u}(n)$. It can then reason as follows

$$\begin{aligned}
\mathsf{plus}(x, \mathsf{i2u}(0)) &= \mathsf{plus}(\mathsf{i2u}(n), \mathsf{i2u}(0)) && (x = \mathsf{i2u}(n)) \\
&= \mathsf{i2u}(n + 0) && (\text{by axiom 3.3}) \\
&= \mathsf{i2u}(n) && (\text{by SMT arithmetic}) \\
&= x && (x = \mathsf{i2u}(n))
\end{aligned}$$

and conclude that the assertion is unsatisfiable, hence the original proof obligation is valid.

3.4. Encoding 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 (2.10), but nothing can be concluded otherwise. For example, the formula

$$f = [x \in \{1, 2, 3\} \mapsto x * x] \Rightarrow f[0] < f[0] + 1, \tag{3.4}$$

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 expression x by two distinct first-order terms depending on whether the *domain condition* $x \in \text{DOMAIN } f$ holds or not: we introduce binary operators α and ω defined as

$$x \in \text{DOMAIN } f \Rightarrow \alpha(f, x) = f[x] \quad \text{and} \quad x \notin \text{DOMAIN } f \Rightarrow \omega(f, x) = f[x].$$

From these conditional definitions, we can derive the theorem

$$f[x] = \text{IF } x \in \text{DOMAIN } f \text{ THEN } \alpha(f, x) \text{ ELSE } \omega(f, x) \tag{3.5}$$

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.

Using theorem (3.5), the expression $f[0]$ in the above example (3.4) is encoded as $\text{IF } 0 \in \text{DOMAIN } f \text{ THEN } \alpha(f, 0) \text{ ELSE } \omega(f, 0)$. The solver would have to use the hypothesis to deduce that $\text{DOMAIN } f = \{1, 2, 3\}$, reducing the condition $0 \in \text{DOMAIN } f$ to false. The conclusion can then be simplified to the

formula $\omega(f,0) < \omega(f,0)+1$, 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 (3.5). From (2.10) and (3.5), we deduce the rewriting rule

$$[x \in S \mapsto e(x)][a] \longrightarrow \text{IF } a \in S \text{ THEN } e(a) \text{ ELSE } \omega([x \in S \mapsto e(x)], a)$$

This rule replaces 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), which is required for termination of $(\text{TLA}^+, \longrightarrow)$ as stated by Theorem 1.

In sorted languages like MS-FOL, there is no notion of function domain other than the types of function arguments. Because explicit functions $[x \in S \mapsto e]$ cannot be mapped directly to first-order expressions, we treat them as any other non-basic expression. The following rewriting rule derived from axiom (2.10) replaces the function construct by a formula containing only basic operators:

$$f = [x \in S \mapsto e] \longrightarrow \mathit{IsAFcn}(f) \land \text{DOMAIN } f = S \land \forall x \in S\colon \alpha(f,x) = e$$

Observe that we have simplified $f[x]$ to $\alpha(f,x)$, because $x \in \text{DOMAIN } f$.

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

$$\begin{array}{ll} \forall f, g\colon & \land\ \mathit{IsAFcn}(f) \land \mathit{IsAFcn}(g) \\ & \land\ \text{DOMAIN } f = \text{DOMAIN } g \\ & \land\ \forall x \in \text{DOMAIN } g\colon \alpha(f,x) = \alpha(g,x) \\ & \Rightarrow f = g \end{array}$$

Again, note that $f[x]$ and $g[x]$ were simplified using α. Unlike set extensionality, this formula is guarded by *IsAFcn*, avoiding the instantiation by expressions that are not considered functions. To prove that DOMAIN f = 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 by adding to the translation an instance of set extensionality for DOMAIN expressions involving expressions that are known to be functions:

$$\begin{array}{ll} \forall f, g\colon & \land\ \mathit{IsAFcn}(f) \land \mathit{IsAFcn}(g) \\ & \land\ \forall x\colon x \in \text{DOMAIN } f \Leftrightarrow x \in \text{DOMAIN } g \\ & \Rightarrow \text{DOMAIN } f = \text{DOMAIN } g \end{array}$$

3.5. Encoding tuples and records

TLA^+ defines n-tuples as functions with domain $1\,..\,n$, and records as functions whose domain is a fixed 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. Below we describe only the encoding of tuples; the encoding of records is analogous.

A tuple $\langle e_1, e_2, \ldots, e_n \rangle$ is defined as the function

$$[i \in 1\,..\,n \mapsto \text{IF } i = 1 \text{ THEN } e_1 \text{ ELSE } (\text{IF } i = 2 \text{ THEN } e_2 \text{ ELSE } (\ \cdots \text{ ELSE } e_n))],$$

so that $\langle e_1, \ldots, e_n \rangle[i] = e_i$ when $i \in 1\,..\,n$. From this definition and from the axioms of extensionality (2.1) and functions (2.10), we derive the rule:

$$\begin{aligned} t = \langle e_1, \ldots, e_n \rangle \longrightarrow\ & \wedge\ \mathit{IsAFcn}(t) \\ & \wedge\ \text{DOMAIN } t = 1\,..\,n \\ & \wedge\ \textstyle\bigwedge_{e_i:\mathsf{U}} \alpha(t, i) = e_i \\ & \wedge\ \textstyle\bigwedge_{e_i:\mathsf{Bool}} \alpha(t, i)^b \Leftrightarrow e_i \end{aligned} \tag{3.6}$$

Note that the translation distinguishes the elements e_i that are Booleans (noted $e_i : \mathsf{Bool}$) from those that are not (noted $e_i : \mathsf{U}$), in line with the Boolification step introduced in Section 3.1.

As in the case of function extensionality, the properties of tuple extensionality are lost when equality is expanded. We need to identify those tuples that have the same structure, that is, those that have the same number of elements. Whenever the rule (3.6) is triggered on a tuple of length n, we add the following formula as an axiom to the translation:

$$\begin{aligned} \forall t_1, t_2 :\ & \wedge\ \mathit{IsAFcn}(t_1) \wedge \mathit{IsAFcn}(t_2) \\ & \wedge\ \text{DOMAIN } t_1 = 1\,..\,n \\ & \wedge\ \text{DOMAIN } t_2 = 1\,..\,n \\ & \wedge\ t_1[1] = t_2[1] \wedge \ldots \wedge t_1[n] = t_2[n] \\ & \Rightarrow\ t_1 = t_2 \end{aligned} \tag{3.7}$$

This approach is limited to tuples whose length can be determined in an obvious way, which is the case in most practical cases. Otherwise, they would be treated as any other function expression.

3.6. Encoding CHOOSE

The CHOOSE operator is not natively supported by automatic provers for first-order logic (FOL or MS-FOL). Nevertheless, we provide support for reasoning about CHOOSE expressions that occur in specifications. By introducing a definition for CHOOSE $x : P(x)$, we obtain the theorem

$$\big(y = \text{CHOOSE } x : P(x)\big) \Rightarrow \big((\exists x : P(x)) \Leftrightarrow P(y)\big),$$

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 (2.9)), suppose an arbitrary pair of CHOOSE expressions $\phi_1 \triangleq \text{CHOOSE } x : P(x)$ and $\phi_2 \triangleq \text{CHOOSE } x : Q(x)$ 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 just need to state the extensionality property for the pair f_1 and f_2 as the axiom $\forall \mathbf{x}, \mathbf{y}\colon \big(\forall z\colon P(z) \Leftrightarrow Q(z)\big) \Rightarrow f_1(\mathbf{x}) = f_2(\mathbf{y})$.

3.7. Encoding strings and IF-THEN-ELSE *expressions*

In FOF, we treat every string as a constant, being careful to avoid name clashes with the variables, and assert that every literal string occurring in the proof obligation is different from each other. In SMT-LIB, strings are encoded using the same technique as arithmetic expressions: for every string literal that occurs in a proof obligation, we declare it as a constant of a newly declared sort Str. Then, we use an injective function $\mathsf{str2u} : \mathsf{Str} \to \mathsf{U}$ to lift string expressions. In TLA$^+$, strings are sequences of characters, and operations such as sequence concatenation can be applied to strings. However, the TLA$^+$ model checker handles strings as constants, and most TLA$^+$ specifications only use constant strings. Similarly, our current implementation has no built-in support for applying sequence operations to strings, but the encoding provides the groundwork for such an extension, which could leverage recent work in SMT solvers on string theories [38].

The expression IF c THEN t ELSE u can be conveniently mapped verbatim using SMT-LIB's conditional operator to $\mathsf{ite}(c, t, u)$, where c is of sort Bool (or boolified), and t and u have the same sort. However, TPTP-FOF does not provide any similar feature, so we have to encode it as a first-order formula.

When both t and u are propositions or Boolified expressions, noted ϕ_1 and ϕ_2, we can apply the equivalence-preserving transformation:

$$\text{IF } c \text{ THEN } \phi_1 \text{ ELSE } \phi_2 \longrightarrow c \Rightarrow \phi_1 \wedge \neg c \Rightarrow \phi_2 \tag{3.8}$$

When t and u are terms, we define the collection of rewriting rules:

$$\mathcal{P}(a, \text{IF } c \text{ THEN } t \text{ ELSE } u) \longrightarrow \text{IF } c \text{ THEN } \mathcal{P}(a, t) \text{ ELSE } \mathcal{P}(a, u) \tag{3.9}$$

where $\mathcal{P}$ is a placeholder for a predicate or an operator parameterized by some other expression a. The formula $\mathcal{P}(a, e)$ can take the form $a = e$, $a \in e$, $a \Rightarrow e$, $a[e]$, etc. For example, when $\mathcal{P}(a, e) \equiv a = e$, the rule becomes

$$a = \text{IF } c \text{ THEN } t \text{ ELSE } u \longrightarrow \text{IF } c \text{ THEN } a = t \text{ ELSE } a = u.$$

These rules are to be read modulo symmetry of the equality symbol. The purpose of the set of rules (3.9) is to distribute $\mathcal{P}$ on the subexpressions t and u while pulling out the IF expression from the non-Boolean operators. Eventually, $\mathcal{P}$ will be a Boolean operator, allowing the application of rule (3.8) or a direct encoding with the ite operator. In fact, we apply the rules (3.9) also during the SMT translation because evidence indicates that they improve its success rate.

This naïve approach to encoding conditional term-expressions could result in an exponential blow-up compared to the size of the original formula, mainly if the sub-expressions are again conditional expressions. The above rules introduce

redundancies: the condition c appears twice in the right-hand side of rule (3.8), and the expression a is also repeated in the rules (3.9). We apply a simple heuristic to abstract the redundant expression. If c or a are just variables, we leave the translation as it is. Otherwise, we abstract the repeated expressions by satisfiability-preserving transformations. For instance, rule (3.8) is modified to introduce a fresh variable z as an abbreviation of c:

$$\text{IF } c \text{ THEN } \phi_1 \text{ ELSE } \phi_2 \longrightarrow \exists z : (z \Leftrightarrow c) \wedge (z \Rightarrow \phi_1) \wedge (\neg z \Rightarrow \phi_2)$$

where $z \Leftrightarrow c$ can be simplified later. This abstraction method is applied analogously to the rules (3.9).

4. Evaluation

In order to validate our approach we took several existing TLAPS proofs, covering verification case studies as well as a standard library containing mathematical theorems about finite sets and their cardinalities. These proofs had been developed interactively using the previously available TLAPS backend provers Zenon, Isabelle/TLA$^+$ and the decision procedure for Presburger arithmetic. We will refer to the combination of those three backends as ZIP for short.

We rewrote the proofs using the new FOL and SMT backend provers. 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 proof obligations generated by the Proof Manager from the proof tree, which 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 2.2GHz Intel Core i7 with 8GB of memory.

Table 2 presents the results for the proofs of four case studies: type correctness and mutual exclusion of the Bakery and Peterson algorithms, three properties about the Memoir security architecture [15], which we divide in type correctness (T) and two refinement proofs (I, A), and type correctness and consistency of the Paxos consensus algorithm.

For each case in the table, we compare how proofs of different sizes are handled by the backends. Each line corresponds to an interactive proof of some given size, that is, the number of generated proof obligations. The third colum gives the ratio of size reduction from the proof in the row below. The next columns correspond to the time an automated prover takes to find a proof for every proof obligation. For each entry in the table, the same prover was executed for all generated proof obligations. The Bakery and Paxos benchmarks require arithmetic reasoning, therefore their entries for the FOL provers are empty. An entry with the symbol "-" means that the solver reached the timeout without finding an automated proof for at least one of the proof obligations. The backends were executed with a timeout of 300 seconds. For our tests we used the off-the-shelf SMT solvers CVC4 v1.3 and Z3 v4.3.2, and the FOL provers E v1.7 and SPASS v3.5.

	size	ratio	ZIP	CVC4	Z3	E	SPASS
Peterson	3	3.3	-	0.41	0.34	63.32	7.55
Peterson	10		5.69	0.78	0.80	4.22	44.64
Bakery	19	11.7	-	36.86	15.20		
Bakery	223		52.74				
Memoir-T	1	12.0	-	-	1.99	-	39.72
Memoir-T	12	35.3	-	3.11	3.21	-	9.43
Memoir-T	424		7.31				
Memoir-I	2	4.5	-	-	-	1.51	1.50
Memoir-I	9	6.8	-	3.84	9.35	3.95	4.11
Memoir-I	62		8.20				
Memoir-A	6	4.5	-	-	-	7.78	12.86
Memoir-A	27	4.7	-	11.31	11.46	9.96	9.99
Memoir-A	127		19.10				
Paxos	83	5.3	-	17.34	15.89		
Paxos	442		36.03				

Table 2: Evaluation results about properties of algorithms and systems.

Table 3 shows results about a set of proofs of theorems about finite sets and their cardinalities. Here we compare the ZIP proofs against a combination of Zenon and SMT solvers, because a few proof obligations generated from big structural high-level formulas can be proved only by Zenon, and the SMT solvers are required for arithmetic reasoning.

In all cases, the use of the new backend leads to significant reductions in interactive proof sizes and running times compared to the original proofs. We consider the proof size to be the more significant metric: the larger the proof, the more interactive proof steps are required, and the more effort the user has to expend for decomposing the proof into steps that the backends can verify. For the algorithms and systems, we observe a reduction in size of about one order of magnitude. In particular, the "shallow" proofs of the first three case

Finite Sets	ZIP		Zenon+SMT		
	size	time	size	time	ratio
CardZero	11	5.42	5	0.48	2.2
CardPlusOne	39	5.35	3	0.49	13.0
CardOne	6	5.36	1	0.35	6.0
CardOneConv	9	0.63	2	0.35	4.5
FiniteSubset	62	7.16	21	5.94	2.9
PigeonHole	42	7.07	20	7.01	2.1
CardMinusOne	11	5.44	5	0.75	2.2

Table 3: Evaluation results for theorems about finite sets and cardinalities.

studies required only minimal interaction. For instance, in the Peterson case, FOL provers and SMT solvers can cope with a proof that generates 3 obligations, whereas the ZIP backends time out in at least one of them. Instead, ZIP requires a more fine-grained proof of size 10. For Finite Sets, the reduction is less impressive, but nevertheless proofs could be reduced to half the original size or better.

The performance of the SMT solvers and the FOL provers is comparable. SMT solvers are very efficient for "shallow" proof obligations, in particular, when few quantifiers need to be instantiated. FOL provers reliably find necessary instantiations for quantified formulas, which may arise from set-theoretic constructions, but they are not appropriate for obligations involving arithmetic reasoning. Beyond our experiments reported here, the SMT backend has been used extensively within TLAPS, and has in fact become the default backend that is invoked before Zenon and Isabelle. It has been instrumental in using TLAPS for developing larger case studies [3, 5].

5. Related work

Some of the encoding techniques described in Section 3, such as rewriting or abstraction, can be considered as simply folklore. Nevertheless, 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, axiomatized TLA^+ functions with domains, including tuples and records, are deeply rooted in the language.

The B and Z specification languages are also based on ZF set theory, although in a somewhat weaker version, because terms and functions have (monomorphic) types in the style of MS-FOL, thus greatly simplifying the translations to SMT languages. Another difference is that functions are defined as binary relations, as is typical in set theory. There are two SMT plugins for the Rodin tool set for Event-B. The SMT solvers plugin [13] directly encodes simple sets (*i.e.*, excluding set of sets) as polymorphic λ-expressions, which are not part of the SMT-LIB standard and are only handled by the parser of the veriT SMT solver. The ppTrans plugin [21] generates different SMT sorts for each combination of simple sets, power sets and cartesian products found in the proof obligation. Therefore, there is one membership operator for every declared set sort, with the advantage that the proof search space is further partitioned, although this requires that the type of every term be known beforehand. (In TLA^+, this can be achieved in certain cases through *type synthesis*; see [29].) Additionally, when ppTrans detects the absence of sets of sets, the translation is further simplified by encoding sets by their characteristic predicates.

Similarly, Atelier-B discharges proof obligations to different SMT solvers through Why3, with similar results to those of Rodin's SMT plug-ins [25]. Set theory, including extensionality, is axiomatized as a new Why3 theory, where sets have an abstract, polymorphic type. The Alt-Ergo SMT solver is particularly useful for Why3 because it natively handles polymorphic first-order formulas [9]. Function application is represented using a binary operator and then

axiomatized. In the context of the BWare project, Conchon et al. [10] proposed many internal optimizations to improve the performance of Alt-Ergo, in order to discharge Atelier-B proof obligations obtained from industrial settings.

ProB includes a translation between TLA$^+$ and B [17], allowing TLA$^+$ users to use ProB tools. It relies on Kodkod, the Alloy Analyzer's backend, to do constraint solving over the first-order fragment of the language, and on the ProB kernel for other expressions [33].

More recently, Delahaye et al. [14] 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 propositions. It is well suited for proof search in axiomatic theories such as Peano arithmetic or Zermelo set thery, as it turns axioms into rewriting rules.

The tool MPTP [36] translates Mizar to TPTP-FOF [34]. The Mizar language, targeted at formalized mathematics, provides 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, it resembles Skolemization.

The abstraction technique that we use for handling non-basic expressions is reminiscent of the Tseitin transformation [35] that introduces new variables for abbreviating sub-formulas when transforming a formula into conjunctive normal form.[9] The abstraction method, just as the Tseitin transformation and also Skolemization, does not preserve logical equivalence because of the additional function symbols it introduces. However, it preserves satisfiability, and this is sufficient for refutation-based theorem proving. Other more sophisticated CNF conversion techniques do not require introducing fresh names for all sub-formulas [4, 12, 31].

The analogue of abstraction in λ-calculus is the λ-lifting transformation, which eliminates free variables from λ-expressions by introducing additional parameters to let-definitions and λ-applications. Thus, bounded let-definitions can be moved out to a global scope. This technique is used to encode Isabelle/HOL functions into MS-FOL [8]. In this encoding, but now comparing to Boolification, formulas and terms are separated by treating all expressions as terms (no predicates are declared in the translation) and injecting expressions expected to be formulas into a new sort isomorphic to Bool. This encoding technique, inspired from Spark [19], is left intentionally incomplete.

6. Conclusions

We have presented a sound and effective way of discharging TLA$^+$ proof obligations using automated theorem provers based on unsorted and many-

[9] The goal of the Tseitin transformation is to avoid the exponential explosion in the number of clauses during clausification, *i.e.* the conversion from a FOL formula to an equi-satisfiable CNF formula.

sorted first-order logic. This encoding forms the core of a back-end prover that integrates external FOL provers and SMT solvers as oracles with the TLA$^+$ Proof System (TLAPS). The main component of the backend is a generic translation framework that makes available to TLAPS any FOL prover or SMT solver that supports the de facto standard formats TPTP-FOF or SMT-LIB.

Our translation enables the backend to successfully encode non-temporal proof obligations in the TLA$^+$ language. The untyped universe of TLA$^+$ is represented as a universal sort. Purely set-theoretic expressions are mapped to formulas over uninterpreted symbols, together with relevant background axioms. For SMT solvers, 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. Functions, tuples, records, and the CHOOSE operator (Hilbert's choice) are encoded using a preprocessing mechanism that combines term rewriting with abstraction. The soundness of the encoding is immediate: all rewriting rules and axioms about sets, functions, records, tuples, *etc.* are theorems in the background theory of TLA$^+$ that have been proved in the Isabelle encoding. 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.

Encouraging results show that FOL provers and 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, in the case of SMT solvers. Both the time required to find automatic proofs and, more importantly, the size of the interactive proof, which reflects the number of user interactions, can be remarkably reduced with our back-end prover.

The translation presented here forms the basis for further optimizations. In [29] 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. What is more important, 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 [8]) the proof objects that many FOL provers and 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.

Acknowledgement. We are grateful to the anonymous reviewers of both this article and the ABZ paper [28] for their comments and suggestions that helped improve the presentation.

References

[1] J.-R. Abrial. Modeling in Event-B – System and Software Engineering, Cambridge University Press, 2010.

[2] J. Avigad. Eliminating Definitions and Skolem Functions in First-order Logic, *ACM Trans. Comput. Logic* 4 (3) (2003) 402–415.

[3] S. Azaiez, D. Doligez, M. Lemerre, T. Libal, S. Merz. Proving Determinacy of the PharOS Real-Time Operating System. *5th Intl. Conf. Abstract State Machines, Alloy, B, TLA, VDM, and Z (ABZ 2016)*. Springer International Publishing, 2016, pp. 70–85.

[4] N. Azmy and C. Weidenbach. Computing tiny clause normal forms. *Proceedings of the 24th International Conference on Automated Deduction*, CADE'13, pages 109–125, 2013. Springer-Verlag.

[5] N. Azmy, S. Merz, C. Weidenbach. A Rigorous Correctness Proof for Pastry. *5th Intl. Conf. Abstract State Machines, Alloy, B, TLA, VDM, and Z (ABZ 2016)*. Springer International Publishing, 2016, pp. 86–101.

[6] F. Baader, T. Nipkow. Term rewriting and all that. Cambridge University Press, 1999.

[7] C. Barrett, A. Stump, C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB), `www.smt-lib.org` (2010).

[8] J. C. Blanchette, S. Böhme, L. C. Paulson. Extending Sledgehammer with SMT solvers. *Journal of Automated Reasoning* 51 (1) (2013) 109–128.

[9] S. Conchon, E. Contejean, J. Kanig, and S. Lescuyer. CC(X): Semantic combination of congruence closure with solvable theories. *Electron. Notes Theor. Comput. Sci.*, 198(2):51–69, May 2008.

[10] S. Conchon and M. Iguernelala. Tuning the Alt-Ergo SMT solver for B proof obligations. In Y. A. Ameur and K.-D. Schewe, editors, *ABZ*, volume 8477 of *Lecture Notes in Computer Science*, pages 294–297. Springer, 2014.

[11] D. Cousineau, D. Doligez, L. Lamport, S. Merz, D. Ricketts, H. Vanzetto. TLA$^+$ Proofs. In D. Giannakopoulou, D. Méry (Eds.), *18th International Symposium on Formal Methods*, Vol. 7436 of LNCS, Springer, 2012, pp. 147–154.

[12] L. de Moura and N. Bjorner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *14th Intl. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*, volume 4963 of *LNCS*, pages 337–340, Budapest, Hungary, 2008. Springer.

[13] D. Déharbe, P. Fontaine, Y. Guyot, L. Voisin. SMT solvers for Rodin. In *3rd Intl. Conf. Abstract State Machines, Alloy, B, VDM, and Z (ABZ 2012)*, Vol. 7316 of LNCS, Springer, Pisa, Italy, 2012, pp. 194–207.

[14] D. Delahaye, D. Doligez, F. Gilbert, P. Halmagrand, O. Hermant. Zenon Modulo: When Achilles Outruns the Tortoise Using Deduction Modulo. In K. McMillan, A. Middeldorp, A. Voronkov (Eds.), *LPAR-19: Logic for Programming, Artificial Intelligence, and Reasoning: 19th International Conference*, Springer, 2013, pp. 274–290.

[15] J. R. Douceur, J. R. Lorch, B. Parno, J. Mickens, J. M. McCune. Memoir—Formal Specs and Correctness Proofs, Tech. Rep. MSR-TR-2011-19, Microsoft Research (2011).

[16] A. Grabowski, A. Kornilowicz, A. Naumowicz. Mizar in a Nutshell, *Journal of Formalized Reasoning* 3 (2) (2010) 152–245.

[17] D. Hansen, M. Leuschel. Translating TLA$^+$ to B for Validation with ProB. In *Proceedings of the 9th Intl. Conference on Integrated Formal Methods*, IFM'12, Springer-Verlag, Berlin, Heidelberg, 2012, pp. 24–38.

[18] G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *J. ACM*, 27(4):797–821, Oct. 1980.

[19] P. B. Jackson and G. O. Passmore. Proving SPARK Verification Conditions with SMT solvers. Dec. 2009.

[20] D. E. Knuth and P. B. Bendix. Simple Word Problems in Universal Algebras. *Computational Problems in Abstract Algebra*, Pergamon, pp. 263–297, 1970.

[21] M. Konrad, L. Voisin. Translation from set-theory to predicate calculus. Technical report, ETH Zurich (2012).

[22] L. Lamport. Specifying Systems: The TLA$^+$ Language and Tools for Hardware and Software Engineers, Addison-Wesley, Boston, Mass., 2002.

[23] L. Lamport, L.C. Paulson. Should your specification language be typed? *ACM Transactions on Programming Languages and Systems (TOPLAS)* Vol. 21 Issue 3, May 1999, pages 502–526.

[24] M. Manzano. Extensions of First-Order Logic, 2nd edition, Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 2005.

[25] D. Mentré, C. Marché, J.-C. Filliâtre, M. Asuka. Discharging Proof Obligations from Atelier B Using Multiple Automated Provers. *Proceedings of the Third International Conference on Abstract State Machines, Alloy, B, VDM, and Z, ABZ'12*, Springer-Verlag, 2012, pp. 238–251.

[26] S. Merz, H. Vanzetto. Automatic Verification of TLA$^+$ Proof Obligations with SMT Solvers. *Proceedings of the 18th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning, LPAR'12*, Springer-Verlag, Berlin, Heidelberg, 2012, pp. 289–303.

[27] S. Merz, H. Vanzetto. Harnessing SMT Solvers for TLA$^+$ Proofs, *Electronic Communications of the European Assoc. of Software, Science and Technology*, Vol. 53.

[28] S. Merz, H. Vanzetto. Encoding TLA$^+$ into Many-Sorted First-Order Logic. In M.J. Butler, K-D. Schewe, A. Mashkoor, M. Bir: *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 5th International Conference (ABZ 2016)*. Springer International Publishing, 2016, pp. 54–69.

[29] S. Merz, H. Vanzetto. Refinement Types for TLA$^+$. In J. M. Badger, K. Y. Rozier (Eds.), *NASA Formal Methods: 6th International Symposium, NFM 2014*, Houston, TX, USA, Springer, 2014, pp. 143–157.

[30] T. Nipkow, L.C. Paulson, M. Wenzel. Isabelle/HOL: a proof assistant for higher-order logic, Vol. 2283 of LNCS, Springer, 2002.

[31] A. Nonnengart, C. Weidenbach. Computing Small Clause Normal Forms. In J.A. Robinson and A. Voronkov (Eds.), *Handbook of Automated Reasoning*, 335–367, Elsevier and MIT Press, 2001.

[32] L. C. Paulson. Set theory for verification: I. from foundations to functions. *Journal of Automated Reasoning*, 11(3):353–389, 1993.

[33] D. Plagge, M. Leuschel. Validating B, Z and TLA$^+$ using ProB and Kodkod. In D. Giannakopoulou, D. Méry (Eds.), *FM 2012: Formal Methods: 18th International Symposium*, Springer, 2012, pp. 372–386.

[34] G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure. *Journal of Automated Reasoning*, 43 (4) (2009) 337–362.

[35] G. S. Tseitin. On the complexity of derivation in propositional calculus. *Studies in constructive mathematics and mathematical logic*, 2(115-125):10–13, 1968.

[36] J. Urban. Translating Mizar for first-order theorem provers. In A. Asperti, B. Buchberger, J. Davenport (Eds.), *Mathematical Knowledge Management*, Vol. 2594 of LNCS, Springer, 2003, pp. 203–215.

[37] C. Weidenbach. SPASS: Combining superposition, sorts and splitting. *Handbook of automated reasoning*, 2:1965–2013, 1999.

[38] Y. Zheng, V. Ganesh, S. Subramanian, O. Tripp, M. Berzish, J. Dolby, X.Zhang. Z3str2: an efficient solver for strings, regular expressions, and length constraints. *Formal Methods in System Design*, 50(2-3):249–288, 2017.

Appendix A. Rewriting rules

This appendix lists the collection of rewriting rules applied during the preprocessing phase of the translation from (boolified) TLA$^+$ to unsorted and many-sorted first-order logic. This list is not comprehensive; trivial rules such as $x \wedge \text{TRUE} \longrightarrow x$ are omitted. All rewriting rules were encoded and mechanically verified in Isabelle/TLA$^+$.

Appendix A.1. First-order logic and CHOOSE *operator*

$$\begin{aligned}
\forall x : x \in \{e_1, \ldots, e_n\} \Rightarrow p(x) &\longrightarrow p(e_1) \wedge \ldots \wedge p(e_n) && (x \notin FV_{1..n}) \\
\exists x : x \in \{e_1, \ldots, e_n\} \wedge p(x) &\longrightarrow p(e_1) \vee \ldots \vee p(e_n) && (x \notin FV_{1..n}) \\
\forall x \in \{y \in S : q(y)\} : p(x) &\longrightarrow \forall x \in S : q(x) \Rightarrow p(x) \\
\exists x \in \{y \in S : q(y)\} : p(x) &\longrightarrow \exists x \in S : q(x) \wedge p(x) \\
y = \text{CHOOSE } x\colon P(x) &\longrightarrow (\exists x\colon P(x)) \Leftrightarrow P(y)
\end{aligned}$$

where $FV_{1..n} = FV(e_1) \cup \ldots \cup FV(e_n)$ and $FV(e)$ is the set of free variables of e.

Appendix A.2. Set theory

$$\begin{aligned}
x \in \{\} &\longrightarrow \text{FALSE} & x \notin S &\longrightarrow \neg(x \in S) \\
x \in \{e_1, \ldots, e_n\} &\longrightarrow x = e_1 \vee \ldots \vee x = e_n & S \subseteq T &\longrightarrow \forall x :\ x \in S \Rightarrow x \in T \\
x \in \{y \in S : p(y)\} &\longrightarrow x \in S \wedge p(x) & x \in e_1 \cup e_2 &\longrightarrow x \in e_1 \vee x \in e_2 \\
S \in \text{SUBSET } T &\longrightarrow \forall x :\ x \in S \Rightarrow x \in T & x \in e_1 \cap e_2 &\longrightarrow x \in e_1 \wedge x \in e_2 \\
x \in \text{UNION } S &\longrightarrow \exists T :\ T \in S \wedge x \in T & x \in e_1 \setminus e_2 &\longrightarrow x \in e_1 \wedge \neg(x \in e_2) \\
x \in e_1 \mathinner{..} e_2 &\longrightarrow x \in \mathit{Int} \wedge e_1 \leq x \wedge x \leq e_2
\end{aligned}$$

Instances of set extensionality:

$$\begin{aligned}
S = \{\} &\longrightarrow \forall x : \neg(x \in S) \\
S = \{e_1, \ldots, e_n\} &\longrightarrow \forall x :\ x \in S \Leftrightarrow x = e_1 \vee \ldots \vee x = e_n \\
S = \text{SUBSET } T &\longrightarrow \forall x :\ x \in S \Leftrightarrow (\forall y :\ y \in x \Rightarrow y \in T) \\
S = \text{UNION } T &\longrightarrow \forall x :\ x \in S \Leftrightarrow (\exists y :\ y \in T \wedge x \in y) \\
S = \{x \in T : p(x)\} &\longrightarrow \forall x :\ x \in S \Leftrightarrow x \in T \wedge p(x) \\
S = \{e(y) : y \in T\} &\longrightarrow \forall x :\ x \in S \Leftrightarrow (\exists y :\ y \in T \wedge x = e(y)) \\
S = T \cup U &\longrightarrow \forall x :\ x \in S \Leftrightarrow x \in T \vee x \in U \\
S = T \cap U &\longrightarrow \forall x :\ x \in S \Leftrightarrow x \in T \wedge x \in U \\
S = T \setminus U &\longrightarrow \forall x :\ x \in S \Leftrightarrow x \in T \wedge \neg(x \in U) \\
\forall x :\ x \in S \Leftrightarrow x \in T &\longrightarrow S = T
\end{aligned}$$

$$
\begin{aligned}
[x \in S \mapsto e(x)][a] &\longrightarrow \text{IF } a \in S \text{ THEN } e(a) \text{ ELSE } \omega([x \in S \mapsto e(x)], a) \\
[f \text{ EXCEPT } ![x] = y][a] &\longrightarrow \text{IF } a \in \text{DOMAIN } f \\
&\qquad \text{THEN } (\text{IF } x = a \text{ THEN } y \text{ ELSE } \alpha(f, a)) \\
&\qquad \text{ELSE } \omega([f \text{ EXCEPT } ![x] = y], a) \\
\text{DOMAIN } [x \in S \mapsto e] &\longrightarrow S \\
\text{DOMAIN } [f \text{ EXCEPT } ![x] = y] &\longrightarrow \text{DOMAIN } f \\
f \in [S \to T] &\longrightarrow \land\ isAFcn(f) \\
&\qquad \land\ \text{DOMAIN } f = S \\
&\qquad \land\ \forall x \in S : \alpha(f, x) \in T \\
[g \text{ EXCEPT } [a] = b] \in [S \to T] &\longrightarrow \land\ isAFcn(g) \\
&\qquad \land\ \text{DOMAIN } g = S \\
&\qquad \land\ a \in S \\
&\qquad \land\ b \in T \\
&\qquad \land\ \forall x \in S \setminus \{a\} : \alpha(f, x) \in T \\
[x \in S' \mapsto e(x)] \in [S \to T] &\longrightarrow \land\ S' = S \\
&\qquad \land\ \forall x \in S : e(x) \in T \\
isAFcn([x \in S \mapsto e]) &\longrightarrow \text{TRUE} \\
isAFcn([f \text{ EXCEPT } ![x] = y]) &\longrightarrow \text{TRUE}
\end{aligned}
$$

Instances of extensionality:

$$
\begin{aligned}
f = [x \in S \mapsto e(x)] &\overset{e(x):\mathsf{Bool}}{\longrightarrow} \land\ isAFcn(f) \\
&\qquad \land\ \text{DOMAIN } f = S \\
&\qquad \land\ \forall x \in S : \alpha(f, x)^b \Leftrightarrow e(x) \\
f = [x \in S \mapsto e(x)] &\longrightarrow \land\ isAFcn(f) \\
&\qquad \land\ \text{DOMAIN } f = S \\
&\qquad \land\ \forall x \in S : \alpha(f, x) = e(x) \\
g = [f \text{ EXCEPT } ![a] = b] &\overset{b:\mathsf{Bool}}{\longrightarrow} \land\ isAFcn(g) \\
&\qquad \land\ \text{DOMAIN } f = \text{DOMAIN } g \\
&\qquad \land\ a \in \text{DOMAIN } g \Rightarrow \alpha(g, a)^b \Leftrightarrow b \\
&\qquad \land\ \forall x \in \text{DOMAIN } f \setminus \{a\} : \alpha(g, x) = \alpha(f, x) \\
g = [f \text{ EXCEPT } ![a] = b] &\longrightarrow \land\ isAFcn(g) \\
&\qquad \land\ \text{DOMAIN } f = \text{DOMAIN } g \\
&\qquad \land\ a \in \text{DOMAIN } g \Rightarrow \alpha(g, a) = b \\
&\qquad \land\ \forall x \in \text{DOMAIN } f \setminus \{a\} : \alpha(f, x) = \alpha(g, x) \\
[x \in S \mapsto e(x)] = [x \in T \mapsto d(x)] &\longrightarrow S = T \land \forall x \in S : e(x) = d(x)
\end{aligned}
$$

Appendix A.4. If-then-else

$$\text{IF } c \text{ THEN } t \text{ ELSE } u \overset{t,u:\mathsf{Bool}}{\longrightarrow} c \Rightarrow t \wedge \neg c \Rightarrow u \qquad \text{(when } c \text{ is a variable)}$$

$$\text{IF } c \text{ THEN } t \text{ ELSE } u \overset{t,u:\mathsf{Bool}}{\longrightarrow} \exists z : (z \Leftrightarrow c) \wedge c \Rightarrow t \wedge \neg c \Rightarrow u$$

$$x \otimes \text{IF } c \text{ THEN } t \text{ ELSE } f \longrightarrow \text{IF } c \text{ THEN } x \otimes t \text{ ELSE } x \otimes f$$

$$f[\text{IF } c \text{ THEN } t \text{ ELSE } u] \longrightarrow \text{IF } c \text{ THEN } f[t] \text{ ELSE } f[u]$$

$$O_1(\text{IF } c \text{ THEN } t \text{ ELSE } u) \longrightarrow \text{IF } c \text{ THEN } O_1(t) \text{ ELSE } O_1(u)$$

where x is a term, $\otimes$ is an infix binary TLA$^+$ operator such as $=$, $\in$, $\Rightarrow$, $\wedge$, $\Leftrightarrow$, $+$, or $<$, and O_1 is a prefix unary TLA$^+$ operator such as $\neg$, DOMAIN, SUBSET or UNION.

Appendix A.5. Tuples and records

Notation: the expression $[h_i \mapsto e_i]_{i:1\,..\,n}$ abbreviates $[h_1 \mapsto e_1, \ldots, h_n \mapsto e_n]$ and $[h_i : e_i]_{i:1\,..\,n}$ abbreviates $[h_1 : e_1, \ldots, h_n : e_n]$.

$$\begin{aligned}
\langle e_1, \ldots, e_n \rangle[i] &\longrightarrow e_i \qquad \text{when } i \in 1\,..\,n \\
t \in S_1 \times \ldots \times S_n &\longrightarrow \wedge\ \mathit{isAFcn}(t) \\
&\quad\ \wedge \text{ DOMAIN } t = 1\,..\,n \\
&\quad\ \wedge\ \alpha(t,1) \in S_1 \wedge \ldots \wedge \alpha(t,n) \in S_n
\end{aligned}$$

$$\begin{aligned}
[h_i \mapsto e_i]_{i:1..n}.h_j &\longrightarrow e_j \qquad \text{when } j \in 1\,..\,n \\
[r \text{ EXCEPT } !.h_1 = e].h_2 &\longrightarrow \text{IF } \text{“}\mathsf{h_1}\text{”} = \text{“}\mathsf{h_2}\text{”} \text{ THEN } e \text{ ELSE } r.h_2 \\
r.h &\longrightarrow r[\text{“}\mathsf{h}\text{”}] \\
r \in [h_i : S_i]_{i:1..n} &\longrightarrow \wedge\ \mathit{isAFcn}(r) \\
&\quad\ \wedge \text{ DOMAIN } r = \{\text{“}\mathsf{h_1}\text{”}, \ldots, \text{“}\mathsf{h_n}\text{”}\} \\
&\quad\ \wedge\ \alpha(r, \text{“}\mathsf{h_1}\text{”}) \in S_1 \wedge \ldots \wedge \alpha(r, \text{“}\mathsf{h_n}\text{”}) \in S_n \\
[h_i \mapsto e_i]_{i:1..n} \in [f_j : S_j]_{j:1..m} &\longrightarrow \wedge\ \{\text{“}\mathsf{h_1}\text{”}, \ldots, \text{“}\mathsf{h_n}\text{”}\} = \{\text{“}\mathsf{f_1}\text{”}, \ldots, \text{“}\mathsf{f_m}\text{”}\} \\
&\quad\ \wedge\ \textstyle\bigwedge e_i \in S_j \qquad \text{when } h_i = f_j, i \in 1\,..\,n, j \in 1\,..\,m
\end{aligned}$$

$$\begin{aligned}
\text{DOMAIN } \langle\rangle &\longrightarrow \{\} \\
\text{DOMAIN } [h_i \mapsto e_i]_{i:1..n} &\longrightarrow \{\text{“}\mathsf{h_1}\text{”}, \ldots, \text{“}\mathsf{h_n}\text{”}\} \\
\text{DOMAIN } \langle e_1, \ldots, e_n \rangle &\longrightarrow 1\,..\,n \\
\text{DOMAIN } [r \text{ EXCEPT } !.h = e] &\longrightarrow \text{DOMAIN } r
\end{aligned}$$

Instances of extensionality:

$$
\begin{aligned}
t = \langle e_1, \ldots, e_n \rangle \longrightarrow\ & \wedge\ isAFcn(t) \\
& \wedge\ \text{DOMAIN}\ t = 1..n \\
& \wedge\ \textstyle\bigwedge_{e_i:\mathsf{Bool}} \alpha(t,i)^b \Leftrightarrow e_i \\
& \wedge\ \textstyle\bigwedge_{e_i:\mathsf{U}} \alpha(t,i) = e_i \\
T = S_1 \times \ldots \times S_n \longrightarrow\ & \forall x :\ x \in T \Leftrightarrow \wedge\ isAFcn(x) \\
& \qquad\qquad \wedge\ \text{DOMAIN}\ x = 1..n \\
& \qquad\qquad \wedge\ \alpha(x,1) \in S_1 \wedge \ldots \wedge \alpha(x,n) \in S_n \\
r = [h_i \mapsto e_i]_{i:1..n} \longrightarrow\ & \wedge\ isAFcn(r) \\
& \wedge\ \text{DOMAIN}\ r = \{\text{“}\mathsf{h_1}\text{”}, \ldots, \text{“}\mathsf{h_n}\text{”}\} \\
& \wedge\ \text{“}\mathsf{h_1}\text{”} \in \text{DOMAIN}\ r \wedge \ldots \wedge \text{“}\mathsf{h_n}\text{”} \in \text{DOMAIN}\ r \\
& \wedge\ \textstyle\bigwedge_{e_i:\mathsf{Bool}} \alpha(r, \text{“}\mathsf{h_i}\text{”})^b \Leftrightarrow e_i \\
& \wedge\ \textstyle\bigwedge_{e_i:\mathsf{U}} \alpha(r, \text{“}\mathsf{h_i}\text{”}) = e_i \\
x = [y\ \text{EXCEPT}\ !.h = e] \longrightarrow\ & \wedge\ isAFcn(x) \\
& \wedge\ \text{DOMAIN}\ x = \text{DOMAIN}\ y \\
& \wedge\ \text{“}\mathsf{h}\text{”} \in \text{DOMAIN}\ y \Rightarrow \alpha(x, \text{“}\mathsf{h}\text{”}) = e \\
& \wedge\ \forall k \in \text{DOMAIN}\ y \setminus \{\text{“}\mathsf{h}\text{”}\} :\ \alpha(x,k) = \alpha(y,k) \\
R = [h_i : S_i]_{i:1..n} \longrightarrow\ & \forall r :\ r \in R \Leftrightarrow \\
& \qquad \wedge\ isAFcn(r) \\
& \qquad \wedge\ \text{DOMAIN}\ r = \{\text{“}\mathsf{h_1}\text{”}, \ldots, \text{“}\mathsf{h_n}\text{”}\} \\
& \qquad \wedge\ \text{“}\mathsf{h_1}\text{”} \in \text{DOMAIN}\ r \wedge \ldots \wedge \text{“}\mathsf{h_n}\text{”} \in \text{DOMAIN}\ r \\
& \qquad \wedge\ \alpha(r, \text{“}\mathsf{h_1}\text{”}) \in S_1 \wedge \cdots \wedge \alpha(r, \text{“}\mathsf{h_n}\text{”}) \in S_n
\end{aligned}
$$