Improving Automation for Higher-Order Proof Steps

Antoine Defourné

Improving Automation for Higher-Order Proof Steps

Antoine Defourné

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

Abstract. We have extended the TLA$^+$ proof system TLAPS with a new backend to improve the automation of proof steps that involve higher-order reasoning. The current support for such steps is poor, requiring the user to break down proofs into unnecessarily small steps. We defined a translation from TLA$^+$ to THF, the TPTP dialect for higher-order logic, and evaluated several higher-order solvers on proof obligations generated from the standard library of TLA$^+$. Our results demonstrate that the solvers are able to handle much coarser proof steps than the other strategies provided by TLAPS, reducing the amount of necessary user interactions by a significant margin.

Keywords: Automated Deduction · Higher-Order Theorem Proving · TLA$^+$ · TLAPS

1 Introduction

TLA$^+$ is a specification language for modelling and expressing properties of distributed systems [10]. Its core logic is the Temporal Logic of Actions (TLA) with the operators and axioms of ZF set theory. The language admits a syntax for expressing theorems and proofs in the hierarchical style of Leslie Lamport [9]. These proofs are treated by the TLA$^+$ Proof System (TLAPS) tool, which dispatches the proof obligations it generates to an array of external solvers, among which there are Isabelle, Zenon, and SMT solvers such as Z3 and CVC4 [6].

This article is about a particular problem that impacts the experience of TLAPS users when writing proofs for TLA$^+$. The language is often categorized as a first-order logic, but in the context of TLAPS, there are situations in which a proof obligation cannot be directly expressed without second-order features. More precisely, a lemma can be parameterized by a first-order operator; when such a lemma is invoked as part of a proof, a second-order unification is necessary. To dispatch obligations to the available backends, TLAPS uses encodings which were almost all designed with first-order logic as a target language. The exception is Isabelle, which means only this solver is currently invoked on the higher-order obligations of TLA$^+$.

Although such higher-order obligations are not the primary kind of obligations one encounters in TLAPS, they are mandatory in some contexts, notably when any form of reasoning by induction is involved. The current support of these obligations by TLAPS is poor: the generic tactics of Isabelle are often unable to handle mildly complex obligations, which forces the user to break down its proofs into smaller steps until they are simple enough for Isabelle. This is a time-consuming process that we want to avoid.

We saw this issue as an opportunity to experiment with a higher-order solver on some TLA$^+$ proofs. Zipperposition was our initial choice for this experiment. It is a superposition theorem prover for first-order logic with equality and theories, recently extended with support for higher-order logic [3, 16, 17], and the winner of the 10th CASC competition (2020) in the THF category [14]. Our main contribution is the implementation of a translation from TLA$^+$ to THF, and the integration of Zipperposition as a new backend for TLAPS. In this article, we also consider other solvers that performed well in the THF division of CASC: Satallax [5], Leo-III [13], Vampire [4] and CVC4 [2]. We evaluate these solvers along with Zipperposition on the same THF problems that TLAPS generates.

The rest of this paper is outlined as follows: in section 2, we present a TLA$^+$ proof that illustrates the problem in more details; in section 3, we present the relevant aspects of the encoding into THF that was implemented; in section 4, we evaluate the solvers and measure how much proof steps we can remove from the original proofs, compared to what is possible with only Isabelle.

2 The Difficulty of Second-order Proofs in TLA$^+$

The difficulties we are interested in arise mostly when dealing with induction in some way. Our example is about defining an operator by recursion in TLA$^+$. We define the operator Sum to represent finite sums over series. The term $\mathsf{Sum}(n, S)$ represents $\sum_{i=1}^{n} S(i)$, where S is any TLA$^+$ first-order operator. The standard library provides a module NaturalsInduction with facilities to deal with such definitions, and a guideline example. Following these guidelines, this is the definition we obtain:

```
sumF(S(_)) ==
    LET sumRec[ m ∈ Nat ] ==
        IF m = 0 THEN 0 ELSE S(m) + sumRec[m - 1]
    IN
    sumRec

Sum(n, S(_)) == sumF(S)[n]
```

Before this definition can be used, a few lemmas need to be proven. The first lemma expresses the fact that a recursive function that matches the definition does exist:

```
THEOREM SumDefConclusion ==
    ASSUME NEW S(_)
    PROVE NatInductiveDefConclusion(sumF(S), 0, LAMBDA v,n : S(n) + v)
```

TLA$^+$ theorems are typically expressed in this manner. The keyword "Assume" precedes a list of declarations and hypotheses, separated by commas. Declarations are introduced by "New", here the only declaration is of an operator S. The keyword "Prove" precedes the actual goal. The content of NatInductiveDefConclusion and the proof of this lemma (omitted here) are not relevant to us. The next lemma reads:

```
THEOREM SumDef ==
    ASSUME NEW S(_), NEW n ∈ Nat
    PROVE Sum(n, S) = IF n = 0 THEN 0 ELSE S(n) + Sum(n - 1, S)
    BY SumDefConclusion DEF NatInductiveDefConclusion, Sum
```

This time we have included the proof, which consists of a single line. The keyword "By" is followed by a list of proven facts to be invoked as hypotheses. "Def" is followed by a list of defined identifiers to expand (by default, operators declared at the top level are not expanded). Here it suffices to invoke the previous lemma, SumDefConclusion, and expand two definitions. As SumDefConclusion is parameterized by an operator S, the resulting obligation for SumDef is higher-order. Unfortunately, Isabelle is unable to solve it.

Inspecting the obligation, we notice that two higher-order instantiations are in fact needed: one for the lemma, the other to instantiate an axiom schema regarding functional application. The usual way to go around such issues is to make an intermediate step to isolate each difficult instantiation:

```
THEOREM SumDef ==
    ASSUME NEW S(_),
           NEW n ∈ Nat
    PROVE Sum(n, S) = IF n = 0 THEN 0 ELSE S(n) + Sum(n - 1, S)
<1>1 NatInductiveDefConclusion(sumF(S), 0, LAMBDA v,m : S(m) + v)
    BY SumDefConclusion
<1> QED
    BY <1>1 DEF NatInductiveDefConclusion, Sum
```

We have replaced the single-line proof by a two-steps proof. The intermediary step is labelled "<1>1", the last step's goal is necessarily "QED" to refer to the main goal. Each step must be justified by its own proof. Note that step <1>1 is invoked as a proved fact for the last step. This time, Isabelle manages to solve <1>1, but the final step is still too difficult. A potential reason for this is that the required instance, sum(S), is too complex a term. We rearrange the proof so that this term is a constant instead, inserting yet another step plus a local definition. This time, Isabelle finishes the proof:

```
THEOREM SumDef ==
    ASSUME NEW S(_),
           NEW n ∈ Nat
    PROVE Sum(n, S) = IF n = 0 THEN 0 ELSE S(n) + Sum(n - 1, S)
<1> DEFINE f == sumF(S)
<1>1 SUFFICES f[n] = IF n = 0 THEN 0 ELSE S(n) + f[n - 1]
    BY DEF Sum
<1>2 NatInductiveDefConclusion(f, 0, LAMBDA v,m : S(m) + v)
```

```
    BY SumDefConclusion
<1> HIDE DEF f
<1> QED
    BY <1>2 DEF NatInductiveDefConclusion
```

The line starting with "DEFINE" is not an intermediary step, but a local definition. Operators declared locally are expanded by default. This is why we also insert the other special line starting with "HIDE DEF": this command makes the operator opaque for the rest of the proof. The new step `<1>1` is an intermediary step that starts with the keyword "SUFFICES". Such steps are used for backward reasoning: instead of proving a new fact, they prove that the current goal can be reformulated.

Not only have we gone from a single-line proof to an over-detailed script, we have also lost time figuring out what made the obligations too complex for Isabelle, and trying different ways to formulate the proof.

Proofs such as this one are typical of the difficult higher-order obligations of TLAPS. Working on these, the user is essentially wasting time trying to find a formulation that accommodates the solvers. This very case is especially problematic, since it was written in accordance with the guidelines provided at the bottom of the module NaturalsInduction—thus it is representative of TLAPS proofs. It would be desirable that the first version, a single-line proof, be handled by TLAPS, and after adding support for Zipperposition we found that the new backend could solve it. Before we develop on the performances of Zipperposition and other higher-order solvers, let us detail how the encoding to THF was implemented.

3 The Encoding of TLA$^+$ into THF

3.1 Overview

Each TLA$^+$ obligation must be encoded into the input language of the relevant backends. We encode the obligations into THF, the component of the TPTP standard for representing problems of higher-order logic.

We ignore the temporal aspects of the logic and view TLA$^+$ as an untyped second-order logic with a standard theory on top (of sets, arithmetic, etc.) [10]. The only logical aspect of TLA$^+$ that sets it apart from traditional logic is the absence of the term/formula distinction. This will be discussed in the next subsection. Although the implementation is new, it is largely inspired by the SMT encoding of TLAPS [15].

The encoding of TLA$^+$ consists in the following sequence of steps:

1. *Disambiguate expressions.* This is where the usual distinction between terms and formulas is recovered.
2. *Apply some elementary simplifications.* Syntactic sugar is removed. Some rewritings are applied to accommodate solvers.

3. *Standardize expressions.* This step only serves a technical purpose. It consists in changing the internal representation of TLA$^+$ primitives. These primitive constructs are rewritten as first- or second-order applications. For instance, the expression $\{x \in S : P(x)\}$ is rewritten as $\mathsf{SetSt}(S, \lambda x.\, P(x))$, where SetSt is a new operator.
4. *Complement the problem with the necessary axioms.* This effectively makes the operators added during the previous step behave like the original constructs. For instance, the axiom of set comprehension will be added to specify SetSt. Only the axioms that specify the operators that occur in the obligation are included.
5. *Translate the problem into THF.* At this point the obligation has been processed enough so that the translation can be direct.

In the next subsections, we give relevant details about the encoding, covering mostly the steps 1 and 2 of this overview.

3.2 Recovering Formulas

First-order logic, even monosorted, still makes a distinction between terms and formulas—we can characterize them with the respective sorts ι and o. The logic of TLA$^+$ can be largely derived from traditional logic, but the distinction between term and formula is absent. All TLA$^+$ expressions belong to the same sort ι, even those that look like formulas.

An important consequence is that any expression can occur in a context where a formula would normally be expected. The expression "$2 \Rightarrow \neg 5$" is legitimate in TLA$^+$. As a particular case, any expression can be treated as a statement; it is allowed to ask if "$1 + 1$" is true or false, for example. The problem arises more commonly with statements of this kind:

```
ASSUME NEW P(_),
       NEW a
PROVE P(a) => P(a)
```

In the absence of any syntactic indication, P cannot be assumed to denote a predicate, even though it can be treated like one in the goal.

In the very first phase of the encoding, expressions are transformed so that the usual distinction between ι and o is recovered. To define this mapping we need some understanding of the semantics of TLA$^+$, which are described in [10, sec. 16.1.3]. TLAPS follows the so-called liberal interpretation of TLA$^+$, which can be summarized as follows: if some e occurs in a boolean context (*i.e.* it must be evaluated as a formula), then it is treated as $e = \top$. For instance, the expression "$2 \Rightarrow \neg 5$" is interpreted like "$(2 = \top) \Rightarrow \neg(5 = \top)$", which happens to be provable by $2 \neq 5$. The last example becomes:

```
ASSUME NEW P(_),
       NEW a
PROVE ( P(a) = TRUE ) => ( P(a) = TRUE )
```

This principle is implemented by the mappings $[\cdot]^f$ and $[\cdot]^t$, defined below. The target language is first-order logic with the sorts ι and o. Two operators are introduced by this mapping: $\mathsf{from\_bool}$ and tt. The former is an injector of o into ι. The latter is the counterpart of $\top$ in the domain of ι—it is needed because $\top$ is treated as a o in the target language, so rewriting e as "$e = \top$" would result in an equality between a ι and a o.

The functions $[\cdot]^f$ and $[\cdot]^t$ map expressions of TLA$^+$ to formulas and terms (respectively) of first-order logic. Variables are denoted by x, operators are denoted by k.

$$\begin{aligned}
[\bot]^f &\triangleq \bot & [x]^t &\triangleq x \\
[e_1 \Rightarrow e_2]^f &\triangleq [e_1]^f \Rightarrow [e_2]^f & [k(e_1, \ldots, e_n)]^t &\triangleq k([e_1]^t, \ldots, [e_n]^t) \\
[\forall x.\, e]^f &\triangleq \forall x.\ [e]^f & [e]^t &\triangleq \mathsf{from\_bool}([e]^f) \\
[e_1 = e_2]^f &\triangleq [e_1]^t = [e_2]^t & & \\
[e]^f &\triangleq [e]^t = \mathsf{tt} & &
\end{aligned}$$

The rules that introduce $\mathsf{from\_bool}$ and tt are the ones that convert terms and formulas into each other. These conversions rules are applied with lowest priority. The actual implementation also accounts for second-order applications; operator arguments are expected to take inputs from ι and return values from ι. Some optimizations are applied: set membership is specified to be a predicate in TLA$^+$, so $\in$ is given a predicate type and no conversion is applied on membership statements; some constructs expect a predicate argument, for example in "$\{x \in S : P(x)\}$", the function $[\cdot]^f$ is called on "$P(x)$".

The mapping $[\cdot]^f$ is sound in the sense that if $[e]^f$ is valid in FOL, then e is valid in TLA$^+$. The mapping is made complete by enforcing the interpretations of the new operators in the target language. This is done by adding a single axiom to the problem:

$$\mathsf{from\_bool}(\top) = \mathsf{tt} \land \mathsf{from\_bool}(\bot) \neq \mathsf{tt} \tag{B}$$

To summarize: for all TLA$^+$ expression e, e is valid iff $[e]^f$ is satisfied by all models of (B) in FOL. The encoding applies $[\cdot]^f$ to all top expressions and adds the axiom (B) to the problem.

3.3 Arithmetic

TLA$^+$ admits a number of primitive operators and axioms, which constitute its standard theory. The main components of this theory are: set theory, functions, and arithmetic. Our encoding simply makes the necessary standard declarations and axioms explicit in the final THF problem, but arithmetic is treated differently.

The SMT encoding uses special axioms that make use of the sort `Int` of SMT-LIB. That way, specialized reasoning implemented by the solvers for arithmetic can be leveraged. But the version of Zipperposition that supports higher-order reasoning does not also support arithmetic, so it is not possible to replicate the method of the SMT encoding.

However, the purpose of using a higher-order solver for TLA$^+$ is not to solve arithmetical goals. The encoding must only be complete enough for goals that involve higher-order reasoning. Therefore, the theory of arithmetic is discarded for our needs; the encoding merely declares the operators it needs, giving them a generic type according to their arities. That includes declaring constants for each literal number that occurs in the obligation.

We don't expect users to invoke the new backend for obligations that require arithmetical reasoning. However, we inspected goals that Zipperposition would not solve and found that simple arithmetical checks were often mandatory. Checking that some value is a member of Int or Nat is a common case. We chose a few axioms to include in the problem so that these checks can be made and more goals can be proven. Here are the axioms we include in the THF file:

Typing axioms Each literal number (constant that identifies a number) is specified to be a member of the set of integers:

$$0 \in \mathsf{Int} \qquad 1 \in \mathsf{Int} \qquad 2 \in \mathsf{Int} \qquad \cdots$$

There is a typing axiom for almost all arithmetical operators, for instance:

$$\forall z_1, z_2 \in \mathsf{Int}.\, (z_1 + z_2) \in \mathsf{Int}$$

Comparisons to 0 In complement to the typing axioms, we add an axiom for each literal number:

$$\begin{array}{ll} z \leq 0 & \text{if } z \text{ is negative} \\ 0 \leq z & \text{if } z \text{ is positive} \end{array}$$

Only the operator for $\leq$ is declared in the THF file. The other comparisons $\geq$, $<$ and $>$ are rewritten so that only $\leq$ occurs during the simplification phase. This is sound with respects to TLA$^+$ semantics, as these operators are defined from each other this way, even for non-integer values.

Distinct literals For every two distinct literals z_1 and z_2 that we declare, we add the axiom:

$$z_1 \neq z_2$$

3.4 Set Extensionality

Past experiences with the SMT encoding of TLA$^+$ showed that encoding set extensionality by including the axiom hardly ever worked, as it is difficult to determine how this axiom should be instantiated in practice. We followed the example of SMT and omitted the axiom of set extensionality. However, while experimenting with our encoding we found that some obligations required set extensionality to be solved. In order to solve more goals, we decided to partially support the axiom by applying simple rewritings.

The axiom states

$$\forall x, y.\, (\forall z.\, z \in x \Leftrightarrow z \in y) \Rightarrow x = y$$

The converse implication is trivially true, so we may apply the axiom as the following rewriting rule:

$$x = y \longrightarrow \forall z.\, z \in x \Leftrightarrow z \in y$$

We determine which equalities are rewritten using polarities. A polarity can be attributed to every subexpression of an expression: the top expression is positive; the polarity is reversed by negation, or by implication for the left member. It is only necessary to rewrite the positive equalities in a goal. These are the equalities that need to be justified, while the negative ones serve as hypotheses and lead to substitutions. For instance, in the goal

$$\forall x, y.\ \{x, y\} = \{y, x\}$$

the equality occurs neither under a negation nor on the left of an implication, so the rewrite rule is applied. However, in the goal

$$\forall x, y.\, x = \emptyset \Rightarrow y \notin x$$

the equality occurs on the left of an implication. No rewriting is applied: the goal is proven by substituting $\emptyset$ for x on the right.

TLA$^+$ is untyped and considers any object to be a set. That means set extensionality is always applicable, and the rewriting rule is sound in every context. But we would not want to rewrite an equality like “$0 = 1$”, for instance. The rule is restricted to cases where one of the members of the equality is built from a set-theoretical primitive (set enumeration, set comprehension, etc.)

This approach is obviously incomplete. Set extensionality may be needed while there is not an equality to rewrite in the goal. For example, this goal cannot be proven with our method:

```
ASSUME NEW F(_), NEW S
PROVE F(S \cup {}) = F(S)
```

Our treatments of set extensionality and arithmetic are the prime sources of incompleteness in the encoding. We do not see this as a problem in the case of arithmetic, as this backend is not intended to be called on arithmetical goals. The lack of a complete support for set extensionality is more often a problem, as it can be natural to write goals that require it when reasoning in terms of set.

4 Evaluation

4.1 Proof Simplification

Having defined and implemented an encoding of TLA$^+$ into THF, we now turn to the evaluation of the higher-order solvers. The purpose of this evaluation is to show that proofs can be written with less details using a higher-order solver, compared to what is currently possible with only Isabelle. Our method consists in evaluating the solvers on simpler versions of existing proofs. Here by "simpler" we mean: proofs carried out in fewer steps. As the proofs get simpler, the resulting obligations must get more complex for the backends.

The general method we applied to evaluate a given solver can be outlined as follows:

1. Select specifications that feature higher-order obligations and identify said obligations;
2. Test the higher-order solver on those obligations, and the surrounding obligations as well;
3. Simplify proofs by merging proof steps around the relevant obligations;
4. Evaluate the solver and Isabelle on the new obligations—we are mostly interested in the number of goals that are uniquely handled by the higher-order solver.

The specifications were selected from the standard library of TLA$^+$. An obligation was considered higher-order if it featured one fact that is parameterized by an operator (proving it requires higher-order unification). Before we show the results, let us briefly explain how step 3 was carried out.

Consider the following proof:

```
THEOREM TailInductiveDef ==
  ASSUME NEW S, NEW Def(_,_), NEW f, NEW f0,
         TailInductiveDefHypothesis(f, S, f0, Def)
  PROVE TailInductiveDefConclusion(f, S, f0, Def)
<1>. DEFINE Op(h,s) == IF s = <<>> THEN f0 ELSE Def(h[Tail(s)], s)
<1>1. StrictSuffixesDetermineDef(S, Op)
  (* ... *)
<1>2. OpDefinesFcn(f, Seq(S), Op)
  BY DEF OpDefinesFcn, TailInductiveDefHypothesis
<1>3. WFInductiveDefines(f, Seq(S), Op)
  BY <1>1, <1>2, SuffixRecursiveSequenceFunctionDef
<1>. QED
  BY <1>3 DEF WFInductiveDefines, TailInductiveDefConclusion
```

The higher-order step here is <1>3. Indeed, the statement of SuffixRecursiveSequenceFunctionDef is:

```
THEOREM SuffixRecursiveSequenceFunctionDef ==
  ASSUME NEW S, NEW Def(_,_), NEW f,
         StrictSuffixesDetermineDef(S, Def),
         OpDefinesFcn(f, Seq(S), Def)
  PROVE WFInductiveDefines(f, Seq(S), Def)
```

The lemma is parameterized by an operator Def(_, _), which makes the obligation associated to <1>3 higher-order. Zipperposition was able to solve that obligation. To find potential simplifications, we test the solver on the surrounding obligations.

Let us assume Zipperposition was able to solve step <1>2 and the QED step, but not <1>1. We can merge <1>3 with <1>2 because the latter is referenced in the proof of the former. We can also merge <1>3 with the QED step, for the same reason. Merging proofs amounts to merging their lists of invoked facts and definitions. After simplification, the result is:

```
THEOREM TailInductiveDef ==
  ASSUME NEW S, NEW Def(_,_), NEW f, NEW f0,
         TailInductiveDefHypothesis(f, S, f0, Def)
  PROVE TailInductiveDefConclusion(f, S, f0, Def)
<1>. DEFINE Op(h,s) == IF s = <<>> THEN f0 ELSE Def(h[Tail(s)], s)
<1>1. StrictSuffixesDetermineDef(S, Op)
  (* ... *)
<1>. QED
  BY <1>1, SuffixRecursiveSequenceFunctionDef
  DEF OpDefinesFcn, TailInductiveDefHypothesis,
      WFInductiveDefines, TailInductiveDefConclusion
```

The new step still contains a reference to <1>1. As Zipperposition was not able to solve this step, it would necessary fail if we merged QED with <1>1, so we stop here. The new proof is 2 steps shorter than the original one, so we measure that simplification by "2 steps". We took Zipperposition as an example, but the same process must be carried out for every solver, resulting in several simplified versions of each specification.

Other simplifications may be applied in particular cases. Some proofs may include inline facts to prove instead of a reference to a lemma or proof step. The proof

```
<1> Cardinality(x \cup {}) = Cardinality(x)
    BY x \cup {} = x, SomeLemma DEF SomeDef
```

is equivalent to

```
<1>1 x \cup {} = x
    OBVIOUS
<1> Cardinality(x \cup {}) = Cardinality(x)
    BY <1>1, SomeLemma DEF SomeDef
```

So the removal of "x \cup {} = x" counts as simplification by one step.

It is also common to find local definitions made opaque for a proof step in order to facilitate unification:

```
<1> DEFINE P(n) == (* ... *)
(* ... *)
<1> HIDE DEF P
<1> ∀ n ∈ Nat : P(n)
    BY <1>1, <1>2, NatInduction
```

We counted as simplification by one step the removal of the Hide command. Removing this line equates to removing the local definition itself, since such a definition is expanded by default.

Table 1. Proof obligations solved by each solver in original specifications

Specification	# Solved higher-order obligations					
	Out of	CVC4	Leo-III	Satallax	Vampire	Zip.
SequenceOpTheorems	18	10	8	10	10	12
FiniteSetTheorems	9	7	0	8	8	8
FunctionTheorems	4	2	1	2	2	3
WellFoundedInduction	8	3	2	4	3	3
Total	**39**	**22**	**11**	**24**	**23**	**26**

4.2 Results

We used TLAPS to generate the necessary Isabelle and THF files from the TLA$^+$ specifications, and then evaluated the backends on these problem files. Isabelle was always tested with the three tactics available in TLAPS: auto, blast and force. The higher-order solvers were evaluated on the same TPTP files. All solvers were run with a timeout of 30 seconds, the default configuration for the Isabelle backend. The experiment was carried out with an Intel Core i7-8650U with 4 cores at 1.90 GHz and 16 GB of RAM. All modules and problem files used for the experiment are publicly available.[1]

The experiment was carried out in two phases, the results of which are summarized in tables 1 and 2. In the first table, we report how many of the original obligations were identified as higher-order and handled by each solver. These obligations come from the standard library of TLA$^+$, so Isabelle necessarily solves all of them.

For each solver, based on the results of that first phase, we edited the specifications by removing a number of proof steps, following the process we described in the previous section. Then, Isabelle and the considered solver were tested on the new specification. In the second table, we report how many proof steps were removed (first column), how many were handled by Isabelle (second column), and how many of the remaining ones were handled by the solver (third column). To compute the results for the last two columns, we searched how many proof steps were removed to obtain each individual obligation. For instance, if one obligation resulted from merging three steps together, it was counted as a reduction by 2 steps. If that obligation was handled by Isabelle or another solved, that would add 2 to its score. An empty cell indicates that the evaluation was not run, because the result could not be other than 0.

4.3 Discussion

On the original specifications (table 1), the performances of all higher-order solvers compare, with only slight variations. Zipperposition solves a bit more obligations, and Leo-III a bit less in general, except for the specification FiniteSetTheorems, on which it did not solve any goal.

[1] https://github.com/adef-inr/Improving-TLAPS-Automation-Frocos-2021.git

Table 2. Proof steps that could be uniquely removed by each HO solver

Solver	Specification	# Proof steps removed		
		Out of	Removed by Isa.	Uniq. removed by solver
CVC4	SequenceOpTheorems	6	0	6
	FiniteSetTheorems	9	2	3
	FunctionTheorems	1	1	–
	WellFoundedInduction	5	1	4
	Total	**21**	**4**	**13**
Leo-III	SequenceOpTheorems	2	0	0
	FiniteSetTheorems	–	–	–
	FunctionTheorems	2	0	0
	WellFoundedInduction	2	0	2
	Total	**6**	**0**	**2**
Satallax	SequenceOpTheorems	4	0	4
	FiniteSetTheorems	10	3	3
	FunctionTheorems	1	1	–
	WellFoundedInduction	4	1	2
	Total	**19**	**5**	**9**
Vampire	SequenceOpTheorems	6	0	4
	FiniteSetTheorems	10	3	3
	FunctionTheorems	1	1	–
	WellFoundedInduction	4	1	3
	Total	**21**	**5**	**10**
Zipperposition	SequenceOpTheorems	19	0	17
	FiniteSetTheorems	10	3	4
	FunctionTheorems	4	1	0
	WellFoundedInduction	4	1	3
	Total	**37**	**5**	**24**

The differences between solvers are more pronounced when we look at the number of proof steps they allowed us to remove (table 2). We could not make much progress with Leo-III, with only 2 steps removed. CVC4, Vampire and Satallax have similar results, with 9–13 proof steps removed. Zipperposition let us remove 24 steps in total, which is significantly higher than any other solver. Out of these 24 steps, 17 come from SequenceOpTheorems, the biggest specification. These 17 steps are shared among only 4 different obligations: two that resulted from the removal of 2 steps each, one from 6 steps, and the last from 7. These are the only cases of obligations resulting from the removal of more than 4 steps. It should also be pointed that we attempted to remove 37 steps with Zipperposition in total. This is higher than for the other solvers, which indicates that Zipperposition could also solve the steps around a higher-order step more often.

Overall, higher-order solvers prove to be helpful for proofs with a few intermediary easy steps. It is often possible to remove a few steps with Zipperposition, CVC4, Satallax or Vampire, and in some cases reduce the whole proof to a single line, as is the case for the following proof:

```
THEOREM SuffixRecursiveSequenceFunctionType ==
  ASSUME NEW S, NEW T, NEW Def(_,_), NEW f,
         T # {},
         StrictSuffixesDetermineDef(S, Def),
         WFInductiveDefines(f, Seq(S), Def),
         ∀ g ∈ [Seq(S) -> T], s ∈ Seq(S) : Def(g,s) ∈ T
  PROVE f ∈ [Seq(S) -> T]
<1>1. IsWellFoundedOn(OpToRel(IsStrictSuffix, Seq(S)), Seq(S))
  BY IsStrictSuffixWellFounded
<1>2. WFDefOn(OpToRel(IsStrictSuffix, Seq(S)), Seq(S), Def)
  BY StrictSuffixesDetermineDef_WFDefOn
<1>. QED
  BY <1>1, <1>2, WFInductiveDefType
```

An important portion of the original obligations are the application of some induction principle. They are variations of this pattern:

```
<1> DEFINE P(n) == (* ... *)
<1>1 P(0)
<1>2 ∀ n ∈ Nat : P(n) => P(n + 1)
<1> HIDE DEF P
<1>3 ∀ n ∈ Nat : P(n) BY <1>1, <1>2, NatInduction (* The HO obligation *)
```

The induction can be on a different structure, but the pattern is the same. All solvers tended to fail on these obligations. When they did succeed, we only removed the Hide line, but after doing so the solver would fail on the new obligation. This is most likely due to the fact that NatInduction must be instantiated with an arbitrary expression instead of the constant P, as P is expanded in the goal when Hide is removed. In some cases, however, Isabelle was able to handle the new obligation—these cases constitute the majority of removed steps that are reported in the second column of table 2. They are especially prevalent in FiniteSetTheorems, as they represent 8 out of the 9 original obligations. This may be the main reason for Leo-III's poor performances on that particular specification.

5 Conclusion

Motivated by the most recent advances in higher-order theorem proving, and the poor support for higher-order proof steps in TLAPS, we implemented an encoding of TLA$^+$ into THF and evaluated several higher-order solvers on a range of proof obligations. Our experiment demonstrated that higher-order solvers are indeed able to handle obligations more complex than TLAPS currently does. Zipperposition in particular outperformed the others by a significant margin, and was integrated in TLAPS as a new backend.

This new backend was not intended to be general-purpose for TLA$^+$, but rather specialized in those obligations that involve a bit of higher-order reasoning. Thus the encoding of TLA$^+$ we implemented is very simple and unoptimized. It appears however that we are unable to solve many obligations precisely because the encoding is lacking on some aspects. Our treatment of set extensionality is insufficient for solving goals such as $\mathrm{Card}(S \cup \emptyset) = \mathrm{Card}(S)$. Our support of arithmetic is very limited, as we provide only a few theory axioms to the solvers, and never consider the full theory. It should also be noted that the encoding was primarily designed with Zipperposition in mind, and that we did not fully explore the options that other solvers offer. For instance, we are aware that Vampire features a special rule for instantiating extensionality axioms, and a set of support strategy for dealing with explicit theory axioms [7,12]. CVC4 features a decision procedure for reasoning about finite sets and cardinalities [1]. These are all potential leads for future improvements of TLA$^+$ encodings, including the general-purpose ones.

Acknowledgment I thank Jasmin Blanchette, Pascal Fontaine and Stephan Merz for their support and guidance through the development of this work. Peter Vukmirović helped with the integration of Zipperposition in TLAPS and its configuration. Simon Cruanes provided additional insights on Zipperposition. Martin Riener tested and helped debugging the new backend. Damien Doligez and Ioannis Filippidis explained to me the inner working of TLAPS, which helped in implementing the extension.

This research is funded by the European Research Council (ERC) under the European's Union Horizon 2020 research and innovation program (grant agreement No. 713999, Matryoshka), and from the Région Grand Est.

References

1. Bansal, K., Reynolds, A., Barrett, C.W., Tinelli, C.: A new decision procedure for finite sets and cardinality constraints in SMT. In: Automated Reasoning - 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27 - July 2, 2016, Proceedings. pp. 82–98 (2016)
2. Barbosa, H., Reynolds, A., Ouraoui, D.E., Tinelli, C., Barrett, C.W.: Extending SMT solvers to higher-order logic. In: Automated Deduction - CADE 27 - 27th International Conference on Automated Deduction, Natal, Brazil, August 27-30, 2019, Proceedings. pp. 35–54 (2019)
3. Bentkamp, A., Blanchette, J., Tourret, S., Vukmirovic, P., Waldmann, U.: Superposition with lambdas. In: Automated Deduction - CADE 27 - 27th International Conference on Automated Deduction, Natal, Brazil, August 27-30, 2019, Proceedings. pp. 55–73 (2019)
4. Bhayat, A., Reger, G.: A combinator-based superposition calculus for higher-order logic. In: Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part I. pp. 278–296 (2020)
5. Brown, C.E.: Satallax: An automatic higher-order prover. In: Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings. pp. 111–117 (2012)

6. Cousineau, D., Doligez, D., Lamport, L., Merz, S., Ricketts, D., Vanzetto, H.: TLA+ Proofs. In: Giannakopoulou, D., Méry, D. (eds.) 18th International Symposium On Formal Methods - FM 2012. Lecture Notes in Computer Science, vol. 7436, pp. 147–154. Springer, Paris, France (Aug 2012), the original publication is available at www.springerlink.com
7. Gupta, A., Kovács, L., Kragl, B., Voronkov, A.: Extensional crisis and proving identity. In: Automated Technology for Verification and Analysis - 12th International Symposium, ATVA 2014, Sydney, NSW, Australia, November 3-7, 2014, Proceedings. pp. 185–200 (2014)
8. Kotelnikov, E., Kovács, L., Voronkov, A.: A first class boolean sort in first-order theorem proving and TPTP. In: Intelligent Computer Mathematics - International Conference, CICM 2015, Washington, DC, USA, July 13-17, 2015, Proceedings. pp. 71–86 (2015)
9. Lamport, L.: How to write a proof. American Mathematical Monthly **102** (September 1995)
10. Lamport, L.: Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley (2002)
11. Mentré, D., Marché, C., Filliâtre, J., Asuka, M.: Discharging proof obligations from atelier B using multiple automated provers. In: Abstract State Machines, Alloy, B, VDM, and Z - Third International Conference, ABZ 2012, Pisa, Italy, June 18-21, 2012. Proceedings. pp. 238–251 (2012)
12. Reger, G., Suda, M.: Set of support for theory reasoning. In: IWIL@LPAR 2017 Workshop and LPAR-21 Short Presentations, Maun, Botswana, May 7-12, 2017 (2017)
13. Steen, A., Benzmüller, C.: The higher-order prover leo-iii. In: Automated Reasoning - 9th International Joint Conference, IJCAR 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings. pp. 108–116 (2018)
14. Sutcliffe, G. (ed.): Proceedings of the 10th IJCAR ATP System Competition (CASC-J10) (July 2020)
15. Vanzetto, H.: Proof automation and type synthesis for set theory in the context of TLA+. (Automatisation de preuves et synthèse de types pour la théorie des ensembles dans le contexte de TLA+). Ph.D. thesis, University of Lorraine, Nancy, France (2014), https://tel.archives-ouvertes.fr/tel-01096518
16. Vukmirovic, P., Bentkamp, A., Nummelin, V.: Efficient full higher-order unification. In: 5th International Conference on Formal Structures for Computation and Deduction, FSCD 2020, June 29-July 6, 2020, Paris, France (Virtual Conference). pp. 5:1–5:17 (2020)
17. Vukmirovic, P., Nummelin, V.: Boolean reasoning in a higher-order superposition prover. In: Joint Proceedings of the 7th Workshop on Practical Aspects of Automated Reasoning (PAAR) and the 5th Satisfiability Checking and Symbolic Computation Workshop (SC-Square) Workshop, 2020 co-located with the 10th International Joint Conference on Automated Reasoning (IJCAR 2020), Paris, France, June-July, 2020 (Virtual). pp. 148–166 (2020)