CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Thesis for the Degree of Licentiate of Engineering

Automated Theorem Proving in a First-Order Logic with First Class Boolean Sort

Evgenii Kotelnikov

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Gothenburg, Sweden
2016

Automated Theorem Proving in a First-Order Logic
with First Class Boolean Sort

Technical Report 152L
ISSN 1652-876X
Department of Computer Science and Engineering

Chalmers University of Technology and
University of Gothenburg

SE-412 96 Gothenburg, Sweden
Telephone +46 (0)31-772 1000

Printed at Reproservice, Chalmers University of Technology
Gothenburg, Sweden, 2016

Abstract

Automated theorem proving is one of the central areas of computer mathematics. It studies methods and techniques for establishing validity of mathematical problems using a computer. The problems are expressed in a variety of formal logics, including first-order logic. Algorithms of automated theorem proving are implemented in computer programs called theorem provers. They find significant application in formal methods of system development and as a mean of automation in proof assistants.

This thesis contributes to automated theorem proving with an extension of many-sorted first-order logic called FOOL. In FOOL boolean sort has a fixed interpretation and boolean terms are treated as formulas. In addition, FOOL contains `if-then-else` and `let-in` constructs. We argue that these extensions are useful for expressing problems coming from program analysis and interactive theorem proving.

We give a formalisation of FOOL and a translation of FOOL formulas to ordinary first-order logic. This translation can be used for proving theorems of FOOL using a first-order theorem prover. We describe our implementation of this translation in the Vampire theorem prover. We extend TPTP, the standard input language of first-order provers, to support formulas of FOOL. We simplify TPTP by providing more powerful and uniform representations of `if-then-else` and `let-in` expressions.

We discuss a modification of superposition calculus that can reason efficiently about formulas with interpreted boolean sort. We present a superposition-friendly translation of FOOL formulas to clausal normal form. We demonstrate usability and high performance of these modifications in Vampire on a series of benchmarks coming from various libraries of problems for automated provers.

Finally, we present an extension of FOOL, aimed to be used for automated program analysis. With this extension, the next state relation of a program can be expressed as a boolean formula which is linear in the size of the program.

Contents

Chapter 1

Introduction

Computer mathematics studies processing of mathematical knowledge with a computer. It explores questions of how to represent mathematical problems and their proofs in a computer, how to check correctness of proofs by a computer and even how to construct proofs automatically using a computer. The latter is the domain of automated theorem proving. It is one of the central and hardest areas of computer mathematics and artificial intelligence. Automated methods of proving theorems precede the existence of computers (see e.g. [15, 17, 23] for a historical survey).

In order to be represented in a computer, a mathematical problem must be expressed in a language of some formal logic. Among the logics used for this purpose are propositional, first-order and higher-order logic, intuitionistic logic, modal, temporal, many-values logic and others.

Algorithms of automated theorem proving are implemented in computer programs called theorem provers. A theorem prover takes a logical conjecture as input and tries to either construct its proof or demonstrate that the conjecture is invalid. Theorem provers can be classified by the logic they support. Propositional, first-order and higher-order logic are among the logics that received the most attention in automated theorem proving. Reasoning in propositional logic, i.e. solving the problem of propositional satisfiability (SAT), is implemented in *SAT solvers*, such as Lingeling [10] and Minisat [40]. Solving the problem of satisfiability modulo theory (SMT) is implemented in *SMT solvers*, such as Z3 [20] and CVC4 [6]. Reasoning is first-order logic is implemented in *first-order theorem provers*, such as Vampire [30], E [39] and iProver [26]. Reasoning in higher-order logic is implemented in *higher-order theorem provers*, such as Satallax [14] and Leo-II [9].

Generally speaking, the more expressive a logic is, the harder it is to reason in it. For example, satisfiability of a propositional problem can always be established, albeit possibly at a high computational cost. Modern SAT solvers implement elaborate algorithms that guarantee good performance characteristics in the average case. In contrast, satisfiability

or unsatisfiability of a first-order problem cannot be in general established by any algorithm. Implementors of first-order provers face the challenge of making the provers succeed on as many real world problems as possible.

Theorem provers are used for software and hardware verification, information management, combinatorial reasoning, and more. They are also the most powerful mean of proof automation in interactive proof assistants. In most applications, the theorem checked by a theorem prover is generated by an external software tool and not given by a human.

This thesis contributes to the area of automated theorem proving by presenting an extension of first-order logic that is useful for applications and can be supported by first-order theorem provers. This chapter describes the background of the thesis and is structured as follows. Section 1.1 gives an introduction to automated theorem proving in first-order logic. Sections 1.2 and 1.3 describe two important applications of first-order provers, automation for program verification and interactive proof assistants. Section 1.4 states the problem addressed in the thesis, and finally Section 1.5 summarises the contributions of the thesis.

1.1 Automated Theorem Proving in First-Order Logic

The problem of establishing validity of a first-order problem automatically can be traced back to Hilbert. In 1928 he posed a question, traditionally referred to as *Entscheidungsproblem*, stated as follows. Is there an algorithm that takes as input a statement in first-order logic and terminates with "Yes" or "No" according to whether or not the statement is valid?

This question has been answered negatively in 1936 independently by Church [16] and Turing [48]. Their proofs rely on the famous Gödel's incompleteness theorem. Gödel's result entails that the problem of provability in first-order logic is not decidable, but *semi-decidable*. Informally, it means that if a logical sentence of the form "φ implies ψ" is valid, that can be established by enumerating all finite derivations in the logical system. A derivation between φ and ψ will necessarily be found in that enumeration. On the other hand, if the sentence is invalid, there is no algorithm that could in general demonstrate that.

The problem of validity of a problem in first-order logic is often formulated in terms of unsatisfiability. Validity of a formula is equivalent to unsatisfiability of its negation. To prove validity of a formula one can derive contradiction from its negation, thus constructing a proof by *refutation*. Conversely, invalidity of a formula is equivalent to satisfiability of its

negation. To demonstrate invalidity of a formula one can find a model of its negation. Algorithms that search for satisfiability and unsatisfiability of first-order formulas are usually implemented as separate procedures.

Despite the complexity of automated reasoning in first-order logic, several methods were found to be efficient for finding unsatisfiability for non-trivial problems. Modern state-or-the-art automated theorem provers are based on superposition calculus [32] and its refinements. Finding satisfiability is a much harder problem because a formula might only have infinite models.

Methods of first-order reasoning usually work not with arbitrary first-order formulas, but with first-order clauses. A first-order formula is in clausal normal form (CNF) if it has the shape $\forall x_1 \ldots \forall x_n (C_1 \wedge \ldots \wedge C_n)$, where $C_i = L_{i,1} \vee \ldots \vee L_{i,l_i}$ and each $L_{i,j}$ is a literal. An alternative representation of a CNF is a set of first-order clauses that form a disjunction. A first-order clause is an implicitly universally quantified disjunction of positive and negative first-order literals. A CNF translation converts an arbitrary first-order formula to CNF, preserving satisfiability. First-order provers that support formulas in full first-order logic implement such translations as part of their preprocessing of the input.

Superposition-based theorem proving stems from the work of Robinson [38] on *resolution calculus*. Resolution calculus establishes unsatisfiability of a set of first-order clauses by systematically and exhaustively applying a set of inference rules which include the resolution inference rule. Resolution calculus is refutationally complete, meaning that a contradiction can be deduced from any unsatisfiable set of clauses. The novelty of Robinson's work was in the usage of *unification* for instantiation of variables. Unification avoids combinatorial explosion of ground instances of quantified formulas that was present e.g. in an earlier algorithm of Davis and Putnam [19][1].

Resolution calculus is refined by *superposition calculus* [2, 3] that employs term orderings for restricting the number of inferences. The basic idea of superposition is to only allow inferences that replace "big" terms by "smaller" ones, with respect to the given ordering.

Most of interesting problems tackled by first-order provers are expressed in first-order logic extended with theories. In order to facilitate superposition reasoning in some theory, it is common to extend superposition calculus with a dedicated inference rule instead of applying standard superposition to an axiomatisation of the theory. For example, instead of axiomatising the equality relation, first-order provers usually extend

[1] This algorithm however retained as the prevalent method for establishing propositional satisfiability after a refinement by Logemann and Loveland [18].

superposition with the paramodulation rule [51, 37]. In a more recent result [22], Vampire was extended to support the extensionality resolution rule to efficiently reason with the extensionality axiom.

First-order provers are currently evaluated on empirical grounds. Comparison between provers are mostly based on success rates and run times on standard corpora of problems. The main corpus is the Thousands of Problems for Theorem Provers (TPTP) library [42]. The problems in this corpus are written in a variety of languages, such as FOF for untyped first-order formulas, TFF0 [45] for typed monomorphic first-order formulas and TFF1 [11] for typed rank-1 polymorphic first-order formulas. The TPTP library is used as a basis for the CASC system competition [46], organised annually.

1.2 Automation for Program Verification

Methods of program verification check that a program satisfies its specification. A program specification can be expressed with logical formulas that annotate program statements, capturing their properties. Typical examples of such properties are pre- and post-conditions, loop invariants and Craig interpolants. These program properties are checked using various tools, including theorem provers (see e.g. [13] for a detailed overview).

Automated program verification sees compliance with specification as a theorem that can be automatically checked by theorem provers. For that, program statements are first translated to logical formulas that capture the semantics of the statements. Then, a theorem is built with the translated formulas as axioms and program properties as the conjecture. Validity of the theorem is interpreted as that the program statements have their annotated properties.

Theorem provers can be used not just for checking program properties, but also for generating them. Recent approaches in interpolation and loop invariant generation [31, 29, 25] present initial results of using first-order theorem provers for generating quantified program properties. First-order theorem provers can also be used to generate program properties with quantifier alternations [29]; such properties could not be generated fully automatically by any previously known method.

1.3 Automation for Proof Assistants

A proof assistant (also called an interactive theorem prover) is a software tool that assists the user in constructing proofs of mathematical problems.

Proof assistants use formalisations of mathematics based on higher-order logic (Isabelle/HOL [33]), type theory (Coq [5]), set theory (Mizar [47]) and others.

Many proof assistants enhance the workflow of their users by automatically filling in parts of the user's proof with the help of tactics. Tactics are specialised scripts that run a predefined collection of proof searching strategies. These strategies can be implemented inside the proof assistant itself or rely on third-party automated theorem provers [12, 49].

To automate proof search for a problem using a theorem prover, a proof assistant first translates the problem into the logic supported by the theorem prover. Since the logics of proof assistants are usually more expressive than the logics of automated provers, the translation can be incomplete. If the theorem prover reports back a proof, the proof assistant uses it to reconstruct a proof in its own logic.

1.4 Problem Statement

Both systems of automated program verification and proof assistants use first-order theorem provers but usually do not natively work with first-order logic. Instead, they translate problems in their respective domains (program properties or formulas in the logic of the proof assistant) to problems in first-order logic. Such translations are not straightforward because of a mismatch between the semantics of first-order logic and that of the domain. They can be very complex and thus error prone.

The performance of a theorem prover on the result of a translation crucially depends on whether the translation introduces formulas potentially making the prover inefficient. Theorem provers, especially first-order ones, are known to be very fragile with respect to the input. Expressing program properties in the "right" format therefore requires solid knowledge about how theorem provers work and are implemented — something that a user of a prover might not have.

If a theorem prover natively supports expressions that mirror the semantics of programming language constructs or features of other logics, we solve both above mentioned problems. First, the users do not have to design translations of such constructs. Second, the users do not have to possess a deep knowledge of how the theorem prover works — the efficiency becomes the responsibility of the prover itself.

This thesis identifies the following syntactical constructs that are generally not supported by first-order provers: first class boolean sort, `if-then-else` expressions and `let-in` expressions. These constructs are

ubiquitous in problems coming from program verification and interactive theorem provers, yet all of them currently require specialised translations. The problem addressed in this thesis is the extension of the input language and underlying logic of first-order theorem provers with these constructs.

Boolean values in programming languages are used both as conditions in conditional or loop statements, and as boolean flags, passed as arguments to functions. A natural way of translating program statements with booleans into formulas is by translating conditions as formulas, and function arguments as terms. Yet we cannot mix boolean terms and formulas in the same way in first-order logic, unless the boolean sort is first class. Properties expressed in higher-order logic routinely use quantification over the interpreted boolean sort; this is not allowed in first-order logic either. Program statements with conditionals can be directly translated to `if-then-else` expressions. Assignments in imperative programs can be translated to a series of nested `let-in` expressions. Both of these constructs are actively used by the users of proof assistants to structure their problems.

1.5 Contributions of the Thesis

This thesis presents an extension of first-order logic that contains the aforementioned missing features of first-order theorem provers and explores how reasoning with this extension can be implemented in existing first-order theorem provers. The extension is called FOOL, standing for first-order logic (FOL) with boolean sort. FOOL differs from the ordinary many-sorted FOL in that it (i) contains an interpreted boolean sort and treats boolean terms as formulas; (ii) supports `if-then-else` expressions; and (iii) supports `let-in` expressions.

The main contributions of the thesis are the following:

1. the definition of FOOL and its semantics;

2. a translation from FOOL formulas to formulas of first-order logic, which can be used to support FOOL in existing first-order provers;

3. a new technique of dealing with the boolean sort in superposition theorem provers that includes replacement of one of the boolean sort axioms with a specialised inference rule;

4. a modification to the TPTP language that makes it compatible with FOOL;

5. an implementation of reasoning in FOOL in the Vampire theorem prover;

6. experimental results showing usability and high performance of the implementation of FOOL in Vampire on a series of benchmarks;

7. a translation of FOOL formulas to first-order clauses that can produce smaller CNFs than the ones obtained by clausification of FOOL formulas translated to first-order logic;

8. an implementation of this translation in Vampire;

9. experimental results showing an increase in performance of Vampire on FOOL problems thanks to the translation of FOOL formulas directly to first-order clauses;

10. a simple extension of FOOL, allowing to express the next state relation of a program as a boolean formula which is linear in the size of the program.

The thesis focuses on practical features extending first-order theorem provers for making them better suited for applications of program verification and proof automation for interactive theorem provers. While the thesis describes implementation details and challenges in the Vampire theorem prover, the described features and their implementation can be carried out in any other first-order prover.

The thesis describes a modification of the TPTP language needed to represent FOOL formulas. This modification will be included in the future TPTP standard.

FOOL contains features of higher-order logic. Some problems that previously required higher-order logic can now be expressed directly in FOOL. For example, the current version of the TPTP library contains over a hundred of such problems. One can check these problems with first-order provers that support FOOL rather than higher-order provers.

FOOL can be regarded as the smallest logic containing both the core theory of SMT-LIB [7] (the standard library of problems for SMT solvers) and the monomorphic first-order part of the TPTP language [45]. First-order provers that support FOOL can therefore reason about some problems from the SMT-LIB library. This opens up an opportunity to evaluate first-order provers on problems that were previously only checked by SMT solvers.

The work described in this thesis has been carried out in three papers, each presented in a separate chapter. Two papers (Chapters 2 and 3)

were published in peer-reviewed conferences, and one (Chapter 4) is currently being prepared for submission to a peer-reviewed conference. The references of the papers have been combined into a single bibliography at the end of the thesis. Other than that, the papers have only been edited for formatting purposes, and in general appear in their original form. The contributions of the thesis are the cumulative contributions of all the papers. The rest of this chapter details the main contributions of each individual paper.

Chapter 2. A First Class Boolean Sort in First-Order Theorem Proving and TPTP

The paper presents the syntax and semantics of FOOL. We show that FOOL is a modification of FOL and reasoning in it reduces to reasoning in FOL. We give a model-preserving translation of FOOL to FOL that can be used for proving theorems in FOOL in a first-order prover. We discuss a modification of superposition calculus that can reason efficiently in the presence of boolean sort. This modification includes replacement of one of the boolean sort axioms with a specialised inference rule that we called FOOL paramodulation. We note that the TPTP language can be changed to support FOOL, which will also simplify some parts of the TPTP syntax.

Statement of contribution. The paper is co-authored with Laura Kovács and Andrei Voronkov. Evgenii Kotelnikov contributed to the formalisation of FOOL and its translation to FOL.

This paper has been published in the proceedings of the 8th Conference on Intelligent Computer Mathematics (CICM) in 2015.

Chapter 3. The Vampire and the FOOL

The paper describes the implementation of FOOL in Vampire. We extend and simplify the TPTP language by providing more powerful and uniform representations of `if-then-else` and `let-in` expressions. We demonstrate usability and high performance of our implementation on two collections of benchmarks, coming from the higher-order part of the TPTP library and from the Isabelle interactive theorem prover. We compare the results of running Vampire on the benchmarks with those of SMT solvers and higher-order provers. Moreover, we compare the performance of Vampire with and without FOOL paramodulation. We give a simple extension

of FOOL, allowing to express the next state relation of a program as a boolean formula which is linear in the size of the program.

Statement of contribution. The paper is co-authored with Laura Kovács, Giles Reger and Andrei Voronkov. Evgenii Kotelnikov contributed with the implementation of FOOL in Vampire and the experiments.

This paper has been published in the proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs (CPP) in 2016.

Chapter 4. A Clausal Normal Form Translation for FOOL

The paper presents a clausification algorithm that translates a FOOL formula to an equisatisfiable set of first-order clauses. This algorithm aims to minimise the number of clauses and the size of the resuting signature, especially on formulas with `if-then-else`, `let-in` expressions and complex boolean structure. We demonstrate by experiments that the implementation of this algorithm in Vampire increases performance of the prover on FOOL problems compared to the earlier translation of FOOL formulas to full first-order logic.

Statement of contribution. The paper is co-written with Martin Suda and based on a joint work together with Laura Kovács and Andrei Voronkov. Evgenii Kotelnikov contributed with the extension of NEW-CNF that supports FOOL, the implementation of this extension in Vampire and the experiments.

Chapter 2

A First Class Boolean Sort in First-Order Theorem Proving and TPTP

Evgenii Kotelnikov, Laura Kovács and Andrei Voronkov

Abstract. To support reasoning about properties of programs operating with boolean values one needs theorem provers to be able to natively deal with the boolean sort. This way, program properties can be translated to first-order logic and theorem provers can be used to prove program properties efficiently. However, in the TPTP language, the input language of automated first-order theorem provers, the use of the boolean sort is limited compared to other sorts, thus hindering the use of first-order theorem provers in program analysis and verification. In this paper, we present an extension FOOL of many-sorted first-order logic, in which the boolean sort is treated as a first-class sort. Boolean terms are indistinguishable from formulas and can appear as arguments to functions. In addition, FOOL contains `if-then-else` and `let-in` constructs. We define the syntax and semantics of FOOL and its model-preserving translation to first-order logic. We also introduce a new technique of dealing with boolean sorts in superposition-based theorem provers. Finally, we discuss how the TPTP language can be changed to support FOOL.

Published in the *Proceedings of the 8th Conference on Intelligent Computer Mathematics*, pages 71–86. Springer, 2015.

2.1 Introduction

Automated program analysis and verification requires discovering and proving program properties. Typical examples of such properties are loop invariants or Craig interpolants. These properties usually are expressed in combined theories of various data structures, such as integers and arrays, and hence require reasoning with both theories and quantifiers. Recent approaches in interpolation and loop invariant generation [31, 29, 25] present initial results of using first-order theorem provers for generating quantified program properties. First-order theorem provers can also be used to generate program properties with quantifier alternations [29]; such properties could not be generated fully automatically by any previously known method. Using first-order theorem prover to generate, and not only prove program properties, opens new directions in analysis and verification of real-life programs.

First-order theorem provers, such as iProver [26], E [39], and Vampire [30], lack however various features that are crucial for program analysis. For example, first-order theorem provers do not yet efficiently handle (combinations of) theories; nevertheless, sound but incomplete theory axiomatisations can be used in a first-order prover even for theories having no finite axiomatisation. Another difficulty in modelling properties arising in program analysis using theorem provers is the gap between the semantics of expressions used in programming languages and expressiveness of the logic used by the theorem prover. A similar gap exists between the language used in presenting mathematics. For example, a standard way to capture assignment in program analysis is to use a `let-in` expression, which introduces a local binding of a variable, or a function for array assignments, to a value. There is no local binding expression in first-order logic, which means that any modelling of imperative programs using first-order theorem provers at the backend, should implement a translation of `let-in` expressions. Similarly, mathematicians commonly use local definitions within definitions and proofs. Some functional programming languages also contain expressions introducing local bindings. In all three cases, to facilitate the use of first-order provers, one needs a theorem prover implementing `let-in` constructs natively.

Efficiency of reasoning-based program analysis largely depends on how programs are translated into a collection of logical formulas capturing the program semantics. The boolean structure of a program property that can be efficiently treated by a theorem prover is however very sensitive to the architecture of the reasoning engine of the prover. Deriving and

expressing program properties in the "right" format therefore requires solid knowledge about how theorem provers work and are implemented — something that a user of a verification tool might not have. Moreover, it can be hard to efficiently reason about certain classes of program properties, unless special inference rules and heuristics are added to the theorem prover, see e.g. [22] when it comes to prove properties of data collections with extensionality axioms.

In order to increase the expressiveness of program properties generated by reasoning-based program analysis, the language of logical formulas accepted by a theorem prover needs to be extended with constructs of programming languages. This way, a straightforward translation of programs into first-order logic can be achieved, thus relieving users from designing translations which can be efficiently treated by the theorem prover. One example of such an extension is recently added to the TPTP language [42] of first-order theorem provers, resembling `if-then-else` and `let-in` expressions that are common in programming languages. Namely, special functions `$ite_t` and `$ite_f` can respectively be used to express a conditional statement on the level of logical terms and formulas, and `$let_tt`, `$let_tf`, `$let_ff` and `$let_ft` can be used to express local variable bindings for all four possible combinations of logical terms (`t`) and formulas (`f`). While satisfiability modulo theory (SMT) solvers, such as Z3 [20] and CVC4 [6], integrate `if-then-else` and `let-in` expressions, in the first-order theorem proving community so far only Vampire supports such expressions.

To illustrate the advantage of using `if-then-else` and `let-in` expressions in automated provers, let us consider the following simple example. We are interested in verifying the partial correctness of the code fragment below:

```
if (r(a)) {
  a := a + 1
} else {
  a := a + q(a)
}
```

using the pre-condition $((\forall x)P(x) \Rightarrow x \geq 0) \wedge ((\forall x)\mathtt{q}(x) > 0) \wedge P(\mathtt{a})$ and the post-condition $\mathtt{a} > 0$. Let `a1` denote the value of the program variable `a` after the execution of the `if`-statement. Using `if-then-else` and `let-in` expressions, the next state function for `a` can naturally be expressed by the following formula:

```
a1 = if r(a) then let a = a + 1 in a
              else let a = a + q(a) in a
```

This formula can further be encoded in TPTP, and hence used by a theorem prover as a hypothesis in proving partial correctness of the above code snippet. We illustrate below the TPTP encoding of the first-order problem corresponding to the partial program correctness problem we consider. Note that the pre-condition becomes a hypothesis in TPTP, whereas the proof obligation given by the post-condition is a TPTP conjecture. All formulas below are typed first-order formulas (`tff`) in TPTP that use the built-in integer sort (`$int`).

```
tff(1, type, p: $int > $o).
tff(2, type, q: $int > $int).
tff(3, type, r: $int > $o).
tff(4, type, a: $int).
tff(5, hypothesis, ![X:$int]: (p(X) => $greatereq(X, 0))).
tff(6, hypothesis, ![X:$int]: ($greatereq(q(X), 0))).
tff(7, hypothesis, p(a)).
tff(8, hypothesis,
    a1 = $ite_t(r(a), $let_tt(a, $sum(a, 1), a),
                      $let_tt(a, $sum(a, q(a)), a))).
tff(9, conjecture, $greater(a1, 0)).
```

Running a theorem prover that supports `$ite_t` and `$let_tt` on this TPTP problem would prove the partial correctness of the program we considered. Note that without the use of `if-then-else` and `let-in` expressions, a more tedious translation is needed for expressing the next state function of the program variable `a` as a first-order formula. When considering more complex programs containing multiple conditional expressions assignments and composition, computing the next state function of a program variable results in a formula of size exponential in the number of conditional expressions. This problem of computing the next state function of variables is well-known in the program analysis community, by computing so-called static single assignment (SSA) forms. Using the `if-then-else` and `let-in` expressions recently introduced in TPTP and already implemented in Vampire [21], one can have a linear-size translation instead.

Let us however note that the usage of conditional expressions in TPTP is somewhat limited. The first argument of `$ite_t` and `$ite_f` is a logical formula, which means that a boolean condition from the program definition should be translated as such. At the same time, the same condition can be treated as a value in the program, for example, in a form of a boolean flag, passed as an argument to a function. Yet we cannot mix terms and formulas in the same way in a logical statement. A possible solution would be to map the boolean type of programs to a user-defined

boolean sort, postulate axioms about its semantics, and manually convert boolean terms into formulas where needed. This approach, however, suffers the disadvantages mentioned earlier, namely the need to design a special translation and its possible inefficiency.

Handling boolean terms as formulas is needed not only in applications of reasoning-based program analysis, but also in various problems of formalisation of mathematics. For example, if one looks at two largest kinds of attempts to formalise mathematics and proofs: those performed by interactive proof assistants, such as Isabelle [33], and the Mizar project [47], one can see that first-order theorem provers are the main workhorses behind computer proofs in both cases – see e.g. [12, 49]. Interactive theorem provers, such as Isabelle routinely use quantifiers over booleans. Let us illustrate this by the following examples, chosen among 490 properties about (co)algebraic datatypes, featuring quantifiers over booleans, generated by Isabelle and kindly found for us by Jasmin Blanchette. Consider the distributivity of a conditional expression (denoted by the *ite* function) over logical connectives, a pattern that is widely used in reasoning about properties of data structures. For lists and the `contains` function that checks that its second argument contains the first one, we have the following example:

$$\begin{array}{l}(\forall p : bool)(\forall l : list_A)(\forall x : A)(\forall y : A)\\ \quad \texttt{contains}(l, \mathrm{ite}(p,\, x,\, y)) \doteq \\ \qquad (p \Rightarrow \texttt{contains}(l,\, x)) \wedge (\neg p \Rightarrow \texttt{contains}(l,\, y))\end{array} \tag{2.1}$$

A more complex example with a heavy use of booleans is the unsatisfiability of the definition of `subset_sorted`.

$$\begin{array}{l}(\forall l_1 : list_A)(\forall l_2 : list_A)(\forall p : bool)\\ \quad \neg(\texttt{subset\_sorted}(l_1,\, l_2) \doteq p \wedge\\ \qquad (\forall l_2' : list_A)\neg(l_1 \doteq \texttt{nil} \wedge l_2 \doteq l_2' \wedge p) \wedge\\ \qquad (\forall x_1 : A)(\forall l_1' : list_A)\neg(l_1 \doteq \texttt{cons}(x_1,\, l_1') \wedge l_2 \doteq \texttt{nil} \wedge \neg p) \wedge\\ \qquad (\forall x_1 : A)(\forall l_1' : list_A)(\forall x_2 : A)(\forall l_2' : list_A)\\ \qquad\quad \neg(l_1 \doteq \texttt{cons}(x_1,\, l_1') \wedge l_2 \doteq \texttt{cons}(x_2,\, l_2') \wedge\\ \qquad\qquad p \doteq \mathrm{ite}(x_1 < x_2,\, \mathit{false},\\ \qquad\qquad\qquad \mathrm{ite}(x_1 \doteq x_2,\\ \qquad\qquad\qquad\quad \texttt{subset\_sorted}(l_1',\, l_2'),\\ \qquad\qquad\qquad\quad \texttt{subset\_sorted}(\texttt{cons}(x_1,\, l_1'),\, l_2')))))\end{array} \tag{2.2}$$

The `subset_sorted` function takes two sorted lists and checks that its second argument is a sublist of the first one.

Problems with boolean terms are also common in the SMT-LIB project [7], the collection of benchmarks for SMT-solvers. Its core logic is a variant of first-order logic that treats boolean terms as formulas, in which logical connectives and conditional expressions are defined in the core theory.

In this paper we propose a modification FOOL of first-order logic, which includes a first-class boolean sort and `if-then-else` and `let-in` expressions, aimed for being used in automated first-order theorem proving. It is the smallest logic that contains both the SMT-LIB core theory and the monomorphic first-order subset of TPTP. The syntax and semantics of the logic are given in Section 2.2. We further describe how FOOL can be translated to the ordinary many-sorted first-order logic in Section 2.3. Section 2.4 discusses superposition-based theorem proving and proposes a new way of dealing with the boolean sort in it. In Section 2.5 we discuss the support of the boolean sort in TPTP and propose changes to it required to support a first-class boolean sort. We point out that such changes can also partially simplify the syntax of TPTP. Section 2.6 discusses related work and Section 2.7 contains concluding remarks.

The main contributions of this paper are the following:

1. the definition of FOOL and its semantics;
2. a translation from FOOL to first-order logic, which can be used to support FOOL in existing first-order theorem provers;
3. a new technique of dealing with the boolean sort in superposition theorem provers, allowing one to replace boolean sort axioms by special rules;
4. a proposal of a change to the TPTP language, intended to support FOOL and also simplify `if-then-else` and `let-in` expressions.

2.2 First-Order Logic with Boolean Sort

First-order logic with the boolean sort (FOOL) extends many-sorted first-order logic (FOL) in two ways:

1. formulas can be treated as terms of the built-in boolean sort; and
2. one can use `if-then-else` and `let-in` expressions defined below.

FOOL is the smallest logic containing both the SMT-LIB core theory and the monomorphic first-order part of the TPTP language. It extends the SMT-LIB core theory by adding `let-in` expressions defining functions and TPTP by the first-class boolean sort.

2.2.1 Syntax

We assume a countable infinite set of *variables*.

Definition 1. A *signature* of first-order logic with the boolean sort is a triple $\Sigma = (S, F, \eta)$, where:

1. S is a set of *sorts*, which contains a special sort *bool*. A *type* is either a sort or a non-empty sequence $\sigma_1, \ldots, \sigma_n, \sigma$ of sorts, written as $\sigma_1 \times \ldots \times \sigma_n \to \sigma$. When $n = 0$, we will simply write σ instead of $\to \sigma$. We call a *type assignment* a mapping from a set of variables and function symbols to types, which maps variables to sorts.

2. F is a set of *function symbols*. We require F to contain binary function symbols $\vee$, $\wedge$, $\Rightarrow$ and $\Leftrightarrow$, used in infix form, a unary function symbol $\neg$, used in prefix form, and nullary function symbols *true*, *false*.

3. η is a *type assignment* which maps each function symbol f into a type τ. When the signature is clear from the context, we will write $f : \tau$ instead of $\eta(f) = \tau$ and say that f is of the type τ.

   We require the symbols $\vee, \wedge, \Rightarrow, \Leftrightarrow$ to be of the type $bool \times bool \to bool$, $\neg$ to be of the type $bool \to bool$ and $true, false$ to be of the type *bool*. ❑

In the sequel we assume that $\Sigma = (S, F, \eta)$ is an arbitrary but fixed signature.

To define the semantics of FOOL, we will have to extend the signature and also assign sorts to variables. Given a type assignment η, we define $\eta, x : \sigma$ to be the type assignment that maps a variable x to σ and coincides otherwise with η. Likewise, we define $\eta, f : \tau$ to be the type assignment that maps a function symbol f to τ and coincides otherwise with η.

Our next aim is to define the set of terms and their sorts with respect to a type assignment η. This will be done using a relation $\eta \vdash t : \sigma$, where $\sigma \in S$, terms can then be defined as all such expressions t.

Definition 2. The relation $\eta \vdash t : \sigma$, where t is an expression and $\sigma \in S$ is defined inductively as follows. If $\eta \vdash t : \sigma$, then we will say that t is a *term of the sort* σ w.r.t. η.

1. If $\eta(x) = \sigma$, then $\eta \vdash x : \sigma$.
2. If $\eta(f) = \sigma_1 \times \ldots \times \sigma_n \to \sigma$, $\eta \vdash t_1 : \sigma_1$, $\ldots$, $\eta \vdash t_n : \sigma_n$, then $\eta \vdash f(t_1, \ldots, t_n) : \sigma$.
3. If $\eta \vdash \varphi : \mathit{bool}$, $\eta \vdash t_1 : \sigma$ and $\eta \vdash t_2 : \sigma$, then $\eta \vdash (\texttt{if } \varphi \texttt{ then } t_1 \texttt{ else } t_2) : \sigma$.
4. Let f be a function symbol and $x_1, \ldots, x_n$ pairwise distinct variables. If $\eta, x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash s : \sigma$ and $\eta, f : (\sigma_1 \times \ldots \times \sigma_n \to \sigma) \vdash t : \tau$, then $\eta \vdash (\texttt{let } f(x_1 : \sigma_1, \ldots, x_n : \sigma_n) = s \texttt{ in } t) : \tau$.
5. If $\eta \vdash s : \sigma$ and $\eta \vdash t : \sigma$, then $\eta \vdash (s \doteq t) : \mathit{bool}$.
6. If $\eta, x : \sigma \vdash \varphi : \mathit{bool}$, then $\eta \vdash (\forall x : \sigma)\varphi : \mathit{bool}$ and $\eta \vdash (\exists x : \sigma)\varphi : \mathit{bool}$. ❑

We only defined a `let-in` expression for a single function symbol. It is not hard to extend it to a `let-in` expression that binds multiple pairwise distinct function symbols in parallel, the details of such an extension are straightforward.

When η is the type assignment function of Σ and $\eta \vdash t : \sigma$, we will say that t is a *Σ-term of the sort* σ, or simply that t is *a term of the sort* σ. It is not hard to argue that every Σ-term has a unique sort.

According to our definition, not every term-like expression has a sort. For example, if x is a variable and η is not defined on x, then x is a not a *term* w.r.t. η. To make the relation between term-like expressions and terms clear, we introduce a notion of free and bound occurrences of variables and function symbols. We call the following occurrences of variables and function symbols *bound*:

1. any occurrence of x in $(\forall x : \sigma)\varphi$ or in $(\exists x : \sigma)\varphi$;
2. in the term $\texttt{let } f(x_1 : \sigma_1, \ldots, x_n : \sigma_n) = s \texttt{ in } t$ any occurrence of a variable x_i in $f(x_1 : \sigma_1, \ldots, x_n : \sigma_n)$ or in s, where $i = 1, \ldots, n$.
3. in the term $\texttt{let } f(x_1 : \sigma_1, \ldots, x_n : \sigma_n) = s \texttt{ in } t$ any occurrence of the function symbol f in $f(x_1 : \sigma_1, \ldots, x_n : \sigma_n)$ or in t.

All other occurrences are called *free*. We say that a variable or a function symbol is *free* in a term t if it has at least one free occurrence in t. A term is called *closed* if it has no occurrences of free variables.

Theorem 1. Suppose $\eta \vdash t : \sigma$. Then

1. for every free variable x of t, η is defined on x;
2. for every free function symbol f of t, η is defined on f;
3. if x is a variable not free in t, and σ' is an arbitrary sort, then $\eta, x : \sigma' \vdash t : \sigma$;
4. if f is a function symbol not free in t, and τ is an arbitrary type, then $\eta, f : \tau \vdash t : \sigma$. ❑

Definition 3. A *predicate symbol* is any function symbol of the type $\sigma_1 \times \ldots \times \sigma_n \to \mathit{bool}$. A Σ-*formula* is a Σ-term of the sort *bool*. All Σ-terms that are not Σ-formulas are called *non-boolean terms*. ❑

Note that, in addition to the use of `let-in` and `if-then-else`, FOOL is a proper extension of first-order logic. For example, in FOOL formulas can be used as arguments to terms and one can quantify over booleans. As a consequence, every quantified boolean formula is a formula in FOOL.

2.2.2 Semantics

As usual, the semantics of FOOL is defined by introducing a notion of *interpretation* and defining how a term is evaluated in an interpretation.

Definition 4. Let η be a type assignment. A η-*interpretation* I is a map, defined as follows. Instead of $I(e)$ we will write $[\![e]\!]_I$, for every element e in the domain of I.

1. Each sort $\sigma \in S$ is mapped to a nonempty domain $[\![\sigma]\!]_I$. We require $[\![\mathit{bool}]\!]_I = \{0, 1\}$.
2. If $\eta \vdash x : \sigma$, then $[\![x]\!]_I \in [\![\sigma]\!]_I$.
3. If $\eta(f) = \sigma_1 \times \ldots \times \sigma_n \to \sigma$, then $[\![f]\!]_I$ is a function from $[\![\sigma_1]\!]_I \times \ldots \times [\![\sigma_n]\!]_I$ to $[\![\sigma]\!]_I$.
4. We require $[\![\mathit{true}]\!]_I = 1$ and $[\![\mathit{false}]\!]_I = 0$. We require $[\![\wedge]\!]_I$, $[\![\vee]\!]_I$, $[\![\Rightarrow]\!]_I$, $[\![\Leftrightarrow]\!]_I$ and $[\![\neg]\!]_I$ respectively to be the logical conjunction, disjunction, implication, equivalence and negation, defined over $\{0, 1\}$ in the standard way. ❑

Given a η-interpretation I and a function symbol f, we define I_f^g to be the mapping that maps f to g and coincides otherwise with I. Likewise, for a variable x and value a we define I_x^a to be the mapping that maps x to a and coincides otherwise with I.

Definition 5. Let I be a η-interpretation, and $\eta \vdash t : \sigma$. The *value of* t *in* I, denoted as $\mathrm{eval}_I(t)$, is a value in $[\![\, \sigma \,]\!]_I$ inductively defined as follows:

$$
\begin{aligned}
\mathrm{eval}_I(x) &= [\![\, x \,]\!]_I\,. \\
\mathrm{eval}_I(f(t_1,\ldots,t_n)) &= [\![\, f \,]\!]_I\,(\mathrm{eval}_I(t_1),\ldots,\mathrm{eval}_I(t_n)). \\
\mathrm{eval}_I(s \doteq t) &= \begin{cases} 1, & \text{if } \mathrm{eval}_I(s) = \mathrm{eval}_I(t); \\ 0, & \text{otherwise.} \end{cases} \\
\mathrm{eval}_I((\forall x : \sigma)\varphi) &= \begin{cases} 1, & \text{if } \mathrm{eval}_{I^a_x}(\varphi) = 1 \\ & \quad \text{for all } a \in [\![\, \sigma \,]\!]_I\,; \\ 0, & \text{otherwise.} \end{cases} \\
\mathrm{eval}_I((\exists x : \sigma)\varphi) &= \begin{cases} 1, & \text{if } \mathrm{eval}_{I^a_x}(\varphi) = 1 \\ & \quad \text{for some } a \in [\![\, \sigma \,]\!]_I\,; \\ 0, & \text{otherwise.} \end{cases} \\
\mathrm{eval}_I(\texttt{if } \varphi \texttt{ then } s \texttt{ else } t) &= \begin{cases} \mathrm{eval}_I(s), & \text{if } \mathrm{eval}_I(\varphi) = 1; \\ \mathrm{eval}_I(t), & \text{otherwise.} \end{cases}
\end{aligned}
$$

$$
\mathrm{eval}_I(\texttt{let } f(x_1 : \sigma_1,\ldots,x_n : \sigma_n) = s \texttt{ in } t) \quad = \quad \mathrm{eval}_{I^g_f}(t),
$$

where g is such that for all $i = 1,\ldots,n$ and $a_i \in [\![\, \sigma_i \,]\!]_I$, we have $g(a_1,\ldots,a_n) = \mathrm{eval}_{I^{a_1\ldots a_n}_{x_1\ldots x_n}}(s)$. ❑

Theorem 2. Let $\eta \vdash \varphi : \mathit{bool}$ and I be a η-interpretation. Then

1. for every free variable x of φ, I is defined on x;
2. for every free function symbol f of φ, I is defined on f;
3. if x is a variable not free in φ, σ is an arbitrary sort, and $a \in [\![\, \sigma \,]\!]_I$ then $\mathrm{eval}_I(\varphi) = \mathrm{eval}_{I^a_x}(\varphi)$;
4. if f is a function symbol not free in φ, $\sigma_1,\ldots,\sigma_n,\sigma$ are arbitrary sorts and $g \in [\![\, \sigma_1 \,]\!]_I \times \ldots \times [\![\, \sigma_n \,]\!]_I \to [\![\, \sigma \,]\!]_I$, then $\mathrm{eval}_I(\varphi) = \mathrm{eval}_{I^g_f}(\varphi)$. ❑

Let $\eta \vdash \varphi : \mathit{bool}$. A η-interpretation I is called a *model* of φ, denoted by $I \models \varphi$, if $\mathrm{eval}_I(\varphi) = 1$. If $I \models \varphi$, we also say that I *satisfies* φ. We say that φ is *valid*, if $I \models \varphi$ for all η-interpretations I, and *satisfiable*, if $I \models \varphi$ for at least one η-interpretation I. Note that Theorem 2 implies that any interpretation, which coincides with I on free variables and free function symbols of φ is also a model of φ.

2.3 Translation of FOOL to FOL

FOOL is a modification of FOL. Every FOL formula is syntactically a FOOL formula and has the same models, but not the other way around. In this section we present a translation from FOOL to FOL, which preserves models. This translation can be used for proving theorems of FOOL using a first-order theorem prover. We do not claim that this translation is efficient – more research is required on designing translations friendly for first-order theorem provers.

We do not formally define many-sorted FOL with equality here, since FOL is essentially a subset of FOOL, which we will discuss now.

We say that an occurrence of a subterm s of the sort *bool* in a term t is in a *formula context* if it is an argument of a logical connective or the occurrence in either $(\forall x : \sigma)s$ or $(\exists x : \sigma)s$. We say that an occurrence of s in t is in a *term context* if this occurrence is an argument of a function symbol, different from a logical connective, or an equality. We say that a formula of FOOL is *syntactically first order* if it contains no `if-then-else` and `let-in` expressions, no variables occurring in a formula context and no formulas occurring in a term context. By restricting the definition of terms to the subset of syntactically first-order formulas, we obtain the standard definition of many-sorted first-order logic, with the only exception of having a distinguished boolean sort and constants *true* and *false* occurring in a formula context.

Let φ be a closed Σ-formula of FOOL. We will perform the following steps to translate φ into a first-order formula. During the translation we will maintain a set of formulas D, which initially is empty. The purpose of D is to collect a set of formulas (definitions of new symbols), which guarantee that the transformation preserves models.

1. Make a sequence of translation steps obtaining a syntactically first order formula φ'. During this translation we will introduce new function symbols and add their types to the type assignment η. We will also add formulas describing properties of these symbols to D. The translation will guarantee that the formulas φ and $\bigwedge_{\psi \in D} \psi \wedge \varphi'$ are equivalent, that is, have the same models restricted to Σ.

2. Replace the constants *true* and *false*, standing in a formula context, by nullary predicates $\top$ and $\bot$ respectively, obtaining a first-order formula.

3. Add special boolean sort axioms.

During the translation, we will say that a function symbol or a variable is *fresh* if it neither appears in φ nor in any of the definitions, nor in the domain of η.

We also need the following definition. Let $\eta \vdash t : \sigma$, and x be a variable occurrence in t. The *sort of this occurrence of* x is defined as follows:

1. any free occurrence of x in a subterm s in the scope of $(\forall x : \sigma')s$ or $(\exists x : \sigma')s$ has the sort σ'.

2. any free occurrence of x_i in a subterm s_1 in the scope of $\texttt{let}\ f(x_1 : \sigma_1, \ldots, x_n : \sigma_n) = s_1\ \texttt{in}\ s_2$ has the sort σ_i, where $i = 1, \ldots, n$.

3. a free occurrence of x in t has the sort $\eta(x)$.

If $\eta \vdash t : \sigma$, s is a subterm of t and x a free variable in s, we say that x has a sort σ' in s if its free occurrences in s have this sort.

The translation steps are defined below. We start with an empty set D and an initial FOOL formula φ, which we would like to change into a syntactically first-order formula. At every translation step we will select a formula χ, which is either φ or a formula in D, which is not syntactically first-order, replace a subterm in χ it by another subterm, and maybe add a formula to D. The translation steps can be applied in any order.

1. Replace a boolean variable x occurring in a formula context, by $x \doteq \mathit{true}$.

2. Suppose that ψ is a formula occurring in a term context such that (i) ψ is different from *true* and *false*, (ii) ψ is not a variable, and (iii) ψ contains no free occurrences of function symbols bound in χ. Let $x_1, \ldots, x_n$ be all free variables of ψ and $\sigma_1, \ldots, \sigma_n$ be their sorts. Take a fresh function symbol g, add the formula $(\forall x_1 : \sigma_1) \ldots (\forall x_n : \sigma_n)(\psi \Leftrightarrow g(x_1, \ldots, x_n) \doteq \mathit{true})$ to D and replace ψ by $g(x_1, \ldots, x_n)$. Finally, change η to $\eta, g : \sigma_1 \times \ldots \times \sigma_n \to \mathit{bool}$.

3. Suppose that $\texttt{if}\ \psi\ \texttt{then}\ s\ \texttt{else}\ t$ is a term containing no free occurrences of function symbols bound in χ. Let $x_1, \ldots, x_n$ be all free variables of this term and $\sigma_1, \ldots, \sigma_n$ be their sorts. Take a fresh function symbol g, add the formulas $(\forall x_1 : \sigma_1) \ldots (\forall x_n : \sigma_n)(\psi \Rightarrow g(x_1, \ldots, x_n) \doteq s)$ and $(\forall x_1 : \sigma_1) \ldots (\forall x_n : \sigma_n)(\neg\psi \Rightarrow g(x_1, \ldots, x_n) \doteq t)$ to D and replace this term by $g(x_1, \ldots, x_n)$. Finally, change η to $\eta, g : \sigma_1 \times \ldots \times \sigma_n \to \sigma_0$, where σ_0 is such that $\eta, x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash s : \sigma_0$.

4. Suppose that $\texttt{let}\ f(x_1 : \sigma_1, \ldots, x_n : \sigma_n) = s\ \texttt{in}\ t$ is a term containing no free occurrences of function symbols bound in χ. Let $y_1, \ldots, y_m$ be all free variables of this term and $\tau_1, \ldots, \tau_m$ be their sorts. Note that the variables in $x_1, \ldots, x_n$ are not necessarily disjoint from the variables in $y_1, \ldots, y_m$.

   Take a fresh function symbol g and fresh sequence of variables $z_1, \ldots, z_n$. Let the term s' be obtained from s by replacing all free occurrences of $x_1, \ldots, x_n$ by $z_1, \ldots, z_n$, respectively. Add the formula $(\forall z_1 : \sigma_1) \ldots (\forall z_n : \sigma_n)(\forall y_1 : \tau_1) \ldots (\forall y_m : \tau_m)(g(z_1, \ldots, z_n, y_1, \ldots, y_m) \doteq s')$ to D. Let the term t' be obtained from t by replacing all bound occurrences of $y_1, \ldots, y_m$ by fresh variables and each application $f(t_1, \ldots, t_n)$ of a free occurrence of f in t by $g(t_1, \ldots, t_n, y_1, \ldots, y_m)$. Then replace $\texttt{let}\ f(x_1 : \sigma_1, \ldots, x_n : \sigma_n) = s\ \texttt{in}\ t$ by t'. Finally, change η to $\eta, g : \sigma_1 \times \ldots \times \sigma_n \times \tau_1 \times \ldots \times \tau_m \to \sigma_0$, where σ_0 is such that $\eta, x_1 : \sigma_1, \ldots, x_n : \sigma_n, y_1 : \tau_1, \ldots, y_m : \tau_m \vdash s : \sigma_0$.

The translation terminates when none of the above rules apply.

We will now formulate several of properties of this translation, which will imply that, in a way, it preserves models. These properties are not hard to prove, we do not include proofs in this paper.

Lemma 1. Suppose that a single step of the translation changes a formula φ_1 into φ_2, δ is the formula added at this step (for step 1 we can assume $\mathit{true} = \mathit{true}$ is added), η is the type assignment before this step and η' is the type assignment after. Then for every η'-interpretation I we have $I \models \delta \Rightarrow (\varphi_1 \Leftrightarrow \varphi_2)$. ❑

By repeated applications of this lemma we obtain the following result.

Lemma 2. Suppose that the translation above changes a formula φ into φ', D is the set of definitions obtained during the translation, η is the initial type assignment and η' is the final type assignment of the translation. Let I' be any interpretation of η'. Then $I' \models \bigwedge_{\psi \in D} \psi \Rightarrow (\varphi \Leftrightarrow \varphi')$. ❑

We also need the following result.

Lemma 3. Any sequence of applications of the translation rules terminates. ❑

The lemmas proved so far imply that the translation terminates and the final formula is equivalent to the initial formula in every interpretation satisfying all definitions in D. To prove model preservation, we also need to prove some properties of the introduced definitions.

Lemma 4. Suppose that one of the steps 2–4 of the translation translates a formula φ_1 into φ_2, δ is the formula added at this step, η is the type assignment before this step, η' is the type assignment after, and g is the fresh function symbol introduced at this step. Let also I be η-interpretation. Then there exists a function h such that $I_g^h \models \delta$. ❑

These properties imply the following result on model preservation.

Theorem 3. Suppose that the translation above translates a formula φ into φ', D is the set of definitions obtained during the translation, η is the initial type assignment and η' is the final type assignment of the translation.

1. Let I be any η-interpretation. Then there is a η'-interpretation I' such that I' is an extension of I and $I' \models \bigwedge_{\psi \in D} \psi \wedge \varphi'$.
2. Let I' be a η'-interpretation and $I' \models \bigwedge_{\psi \in D} \psi \wedge \varphi'$. Then $I' \models \varphi$. ❑

This theorem implies that φ and $\bigwedge_{\psi \in D} \psi \wedge \varphi'$ have the same models, as far as the original type assignment (the type assignment of Σ) is concerned. The formula $\bigwedge_{\psi \in D} \psi \wedge \varphi'$ in this theorem is syntactically first-order. Denote this formula by γ. Our next step is to define a model-preserving translation from syntactically first-order formulas to first-order formulas.

To make γ into a first-order formula, we should get rid of *true* and *false* occurring in a formula context. To preserve the semantics, we should also add axioms for the boolean sort, since in first-order logic all sorts are uninterpreted, while in FOOL the interpretations of the boolean sort and constants *true* and *false* are fixed.

To fix the problem, we will add axioms expressing that the boolean sort has two elements and that *true* and *false* represent the two distinct elements of this sort.

$$\forall (x : \mathit{bool})(x \doteq \mathit{true} \vee x \doteq \mathit{false}) \wedge \mathit{true} \not\doteq \mathit{false}. \tag{2.3}$$

Note that this formula is a tautology in FOOL, but not in FOL.

Given a syntactically first-order formula γ, we denote by $\mathit{fol}(\gamma)$ the formula obtained from γ by replacing all occurrences of *true* and *false* in a formula context by logical constants $\top$ and $\bot$ (interpreted as always true and always false), respectively and adding formula (2.3).

Theorem 4. Let η is a type assignment and γ be a syntactically first-order formula such that $\eta \vdash \gamma : \mathit{bool}$.

1. Suppose that I is a η-interpretation and $I \models \gamma$ in FOOL. Then $I \models \mathit{fol}(\gamma)$ in first-order logic.

2. Suppose that I is a η-interpretation and $I \models \mathit{fol}(\gamma)$ in first-order logic. Consider the FOOL-interpretation I' that is obtained from I by changing the interpretation of the boolean sort *bool* by $\{0, 1\}$ and the interpretations of *true* and *false* by the elements 1 and 0, respectively, of this sort. Then $I' \models \gamma$ in FOOL. ❑

Theorems 3 and 4 show that our translation preserves models. Every model of the original formula can be extended to a model of the translated formulas by adding values of the function symbols introduced during the translation. Likewise, any first-order model of the translated formula becomes a model of the original formula after changing the interpretation of the boolean sort to coincide with its interpretation in FOOL.

2.4 Superposition for FOOL

In Section 2.3 we presented a model-preserving syntactic translation of FOOL to FOL. Based on this translation, automated reasoning about FOOL formulas can be done by translating a FOOL formula into a FOL formula, and using an automated first-order theorem prover on the resulting FOL formula. State-of-the-art first-order theorem provers, such as Vampire [30], E [39] and Spass [50], implement superposition calculus for proving first-order formulas. Naturally, we would like to have a translation exploiting such provers in an efficient manner.

Note however that our translation adds the two-element domain axiom $\forall(x : \mathit{bool})(x \doteq \mathit{true} \vee x \doteq \mathit{false})$ for the boolean sort. This axioms will be converted to the clause

$$x \doteq \mathit{true} \vee x \doteq \mathit{false}, \tag{2.4}$$

where x is a boolean variable. In this section we explain why this axiom requires a special treatment and propose a solution to overcome problems caused by its presence.

We assume some basic understanding of first-order theorem proving and superposition calculus, see, e.g. [4, 32]. We fix a superposition inference system for first-order logic with equality, parametrised by a simplification ordering $\succ$ on literals and a well-behaved literal selection function [30], that is a function that guarantees completeness of the calculus. We denote selected literals by underlining them. We assume that

equality literals are treated by a dedicated inference rule, namely, the ordered paramodulation rule [37]:

$$\frac{\underline{l \doteq r} \vee C \quad \underline{L[s]} \vee D}{(L[r] \vee C \vee D)\theta} \quad \text{if } \theta = \mathrm{mgu}(l, s),$$

where C, D are clauses, L is a literal, l, r, s are terms, $\mathrm{mgu}(l, s)$ is a most general unifier of l and s, and $r\theta \not\succeq l\theta$. The notation $L[s]$ denotes that s is a subterm of L, then $L[r]$ denotes the result of replacement of s by r.

Suppose now that we use an off-the-shelf superposition theorem prover to reason about FOL formulas obtained by our translation. W.l.o.g, we assume that $\mathit{true} \succ \mathit{false}$ in the term ordering used by the prover. Then self-paramodulation (from *true* to *true*) can be applied to clause (2.4) as follows:

$$\frac{\underline{x \doteq \mathit{true}} \vee x \doteq \mathit{false} \quad \underline{y \doteq \mathit{true}} \vee y \doteq \mathit{false}}{x \doteq y \vee x \doteq \mathit{false} \vee y \doteq \mathit{false}}$$

The derived clause $x \doteq y \vee x \doteq \mathit{false} \vee y \doteq \mathit{false}$ is a recipe for disaster, since the literal $x \doteq y$ must be selected and can be used for paramodulation into every non-variable term of a boolean sort. Very soon the search space will contain many clauses obtained as logical consequences of clause (2.4) and results of paramodulation from variables applied to them. This will cause a rapid degradation of performance of superposition-based provers.

To get around this problem, we propose the following solution. First, we will choose term orderings $\succ$ having the following properties: $\mathit{true} \succ \mathit{false}$ and *true* and *false* are the smallest ground terms w.r.t. $\succ$. Consider now all ground instances of (2.4). They have the form $s \doteq \mathit{true} \vee s \doteq \mathit{false}$, where s is a ground term. When s is either *true* or *false*, this instance is a tautology, and hence redundant. Therefore, we should only consider instances for which $s \succ \mathit{true}$. This prevents self-paramodulation of (2.4).

Now the only possible inferences with (2.4) are inferences of the form

$$\frac{\underline{x \doteq \mathit{true}} \vee x \doteq \mathit{false} \quad C[s]}{C[\mathit{true}] \vee s \doteq \mathit{false}} \ ,$$

where s is a non-variable term of the sort *bool*. To implement this, we can remove clause (2.4) and add as an extra inference rule to the superposition calculus the following rule:

$$\frac{C[s]}{C[\mathit{true}] \vee s \doteq \mathit{false}} \ ,$$

where s is a non-variable term of the sort *bool* other than *true* and *false*.

2.5 TPTP support for FOOL

The typed monomorphic first-order formulas subset, called TFF0, of the TPTP language [42], is a representation language for many-sorted first-order logic. It contains `if-then-else` and `let-in` constructs (see below), which is useful for applications, but is inconsistent in its treatment of the boolean sort. It has a predefined atomic sort symbol `$o` denoting the boolean sort. However, unlike all other sort symbols, `$o` can only be used to declare the return type of predicate symbols. This means that one cannot define a function having a boolean argument, use boolean variables or equality between booleans.

Such an inconsistent use of the boolean sort results in having two kinds of `if-then-else` expressions and four kinds of `let-in` expressions. For example, a FOOL-term `let` $f(x_1 : \sigma_1, \ldots, x_n : \sigma_n) = s$ `in` t can be represented using one of the four TPTP alternatives `$let_tt`, `$let_tf`, `$let_ft` or `$let_ff`, depending on whether s and t are terms or formulas.

Since the boolean type is second-class in TPTP, one cannot directly represent formulas coming from program analysis and interactive theorem provers, such as formulas (2.1) and (2.2) of Section 2.1.

We propose to modify the TFF0 language of TPTP to coincide with FOOL. It is not late to do so, since there is no general support for `if-then-else` and `let-in`. To the best of our knowledge, Vampire is currently the only theorem prover supporting full TFF0. Note that such a modification of TPTP would make multiple forms of `if-then-else` and `let-in` redundant. It will also make it possible to directly represent the SMT-LIB core theory.

We note that our changes and modifications on TFF0 can also be applied to the TFF1 language of TPTP [11]. TFF1 is a polymorphic extension of TFF0 and its formalisation does not treat the boolean sort. Extending our work to TFF1 should not be hard but has to be done in detail.

2.6 Related Work

Handling boolean terms as formulas is common in the SMT community. The SMT-LIB project [7] defines its core logic as first-order logic extended with the distinguished first-class boolean sort and the `let-in` expression used for local bindings of variables. The core theory of SMT-LIB defines

logical connectives as boolean functions and the ad-hoc polymorphic `if-then-else` (*ite*) function, used for conditional expressions. The language FOOL defined here extends the SMT-LIB core language with local function definitions, using `let-in` expressions defining functions of arbitrary, and not just zero, arity. This, FOOL contains both this language and the TFF0 subset of TPTP. Further, we present a translation of FOOL to FOL and show how one can improve superposition theorem provers to reason with the boolean sort.

Efficient superposition theorem proving in finite domains, such as the boolean domain, is also discussed in [24]. The approach of [24] sometimes falls back to enumerating instances of a clause by instantiating finite domain variables with all elements of the corresponding domains. We point out here that for the boolean (i.e., two-element) domain there is a simpler solution. However, the approach of [24] also allows one to handle domains with more than two elements. One can also generalise our approach to arbitrary finite domains by using binary encodings of finite domains, however, this will necessarily result in loss of efficiency, since a single variable over a domain with 2^k elements will become k variables in our approach, and similarly for function arguments.

2.7 Conclusion

We defined first-order logic with the first class boolean sort (FOOL). It extends ordinary many-sorted first-order logic (FOL) with (i) the boolean sort such that terms of this sort are indistinguishable from formulas and (ii) `if-then-else` and `let-in` expressions. The semantics of `let-in` expressions in FOOL is essentially their semantics in functional programming languages, when they are not used for recursive definitions. In particular, non-recursive local functions can be defined and function symbols can be bound to a different sort in nested `let-in` expressions.

We argued that these extensions are useful in reasoning about problems coming from program analysis and interactive theorem proving. The extraction of properties from certain program definitions (especially in functional programming languages) into FOOL formulas is more straightforward than into ordinary FOL formulas and potentially more efficient. In a similar way, a more straightforward translation of certain higher-order formulas into FOOL can facilitate proof automation in interactive theorem provers.

FOOL is a modification of FOL and reasoning in it reduces to reasoning in FOL. We gave a translation of FOOL to FOL that can be used

for proving theorems in FOOL in a first-order theorem prover. We further discussed a modification of superposition calculus that can reason efficiently in presence of the boolean sort. Finally, we pointed out that the TPTP language can be changed to support FOOL, which will also simplify some parts of the TPTP syntax.

Implementation of theorem proving support for FOOL, including its superposition-friendly translation to CNF, is an important task for future work. Further, we are also interested in extending FOOL with theories, such as the theory of integer linear arithmetic and arrays.

Acknowledgements

The first two authors were partially supported by the Wallenberg Academy Fellowship 2014, the Swedish VR grant D0497701, and the Austrian research project FWF S11409-N23. The third author was Partially supported by the EPSRC grant “Reasoning in Verification and Security”.

Chapter 3

The Vampire and the FOOL

Evgenii Kotelnikov, Laura Kovács,
Giles Reger and Andrei Voronkov

Abstract. This paper presents new features recently implemented in the theorem prover Vampire, namely support for first-order logic with a first class boolean sort (FOOL) and polymorphic arrays. In addition to having a first class boolean sort, FOOL also contains `if-then-else` and `let-in` expressions. We argue that presented extensions facilitate reasoning-based program analysis, both by increasing the expressivity of first-order reasoners and by gains in efficiency.

Published the *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, pages 37–48. ACM New York, 2016.

3.1 Introduction

Automated program analysis and verification requires discovering and proving program properties. These program properties are checked using various tools, including theorem provers. The translation of program properties into formulas accepted by a theorem prover is not straightforward because of a mismatch between the semantics of the programming language constructs and that of the input language of the theorem prover. If program properties are not directly expressible in the input language, one should implement a translation of such program properties to the language. Such translations can be very complex and thus error prone.

The performance of a theorem prover on the result of a translation crucially depends on whether the translation introduces formulas potentially making the prover inefficient. Theorem provers, especially first-order ones, are known to be very fragile with respect to the input. Expressing program properties in the "right" format therefore requires solid knowledge about how theorem provers work and are implemented — something that a user of a verification tool might not have. Moreover, it can be hard to efficiently reason about certain classes of program properties, unless special inference rules and heuristics are added to the theorem prover. For example, [22] shows a considerable gain in performance on proving properties of data collections by using a specially designed extensionality resolution rule.

If a theorem prover natively supports expressions that mirror the semantics of programming language constructs, we solve both above mentioned problems. First, the users do not have to design translations of such constructs. Second, the users do not have to possess a deep knowledge of how the theorem prover works — the efficiency becomes the responsibility of the prover itself.

In this work we present new features recently developed and implemented in the theorem prover Vampire [30] to natively support mirroring programming language constructs in its input language. They include (i) FOOL [28], that is the extension of first-order logic by a first-class boolean sort, `if-then-else` and `let-in` expressions, and (ii) polymorphic arrays.

This paper is structured as follows. Section 3.2 presents how FOOL is implemented in Vampire and focuses on new extensions to the TPTP input language [42] of first-order provers. Section 3.2 extends the TPTP language of monomorphic many-sorted first-order formulas, called TFF0 [45], and allows users to treat the built-in boolean sort `$o` as a first class sort.

Moreover, it introduces expressions `$ite` and `$let`, which unify various TPTP `if-then-else` and `let-in` expressions.

Section 3.3 presents a formalisation of a polymorphic theory of arrays in TPTP and its implementation in Vampire. It extends TPTP with features of the TFF1 language [11] of rank-1 polymorphic first-order formulas, namely, sort arguments for the built-in array sort constructor `$array`. Sort variables however are not supported.

We argue that these extensions make the translation of properties of some programs to TPTP easier. To support this claim, in Section 3.4 we discuss representation of various programming and other constructs in the extended TPTP language. We also give a linear translation of the next state relation for any program with assignments, `if-then-else`, and sequential composition.

Experiments with theorem proving with FOOL formulas are described in Section 3.5. In particular, we show that the implementation of a new inference rule, called FOOL paramodulation, improves performance of theorem provers using superposition calculus.

Finally, Section 3.6 discusses related work and Section 3.7 outlines future work.

Summary of the main results.

- We describe an implementation of first-order logic with a first-class boolean sort. This bridges the gap between input languages for theorem provers and logics and tools used in program analysis. We believe it is a first ever implementation of first-class boolean sorts in superposition theorem provers.

- We extend and simplify the TPTP language [42], by providing more powerful and more uniform representations of `if-then-else` and `let-in` expressions. To the best of our knowledge, Vampire is the only superposition theorem prover implementing these constructs.

- We formalise and describe an implementation in Vampire of a polymorphic theory of arrays. Again, we believe that Vampire is the only superposition theorem prover implementing this theory.

- We give a simple extension of FOOL, allowing to express the next state relation of a program as a boolean formula which is linear in the size of the program. This boolean formula captures the exact semantics of the program and can be used by a first-order theorem

prover. We are not aware of any other work on extending theorem provers with support for representing fragments of imperative programs.

- We demonstrate usability and high performance of our implementation on two collections of examples, coming from the higher-order part of the TPTP library and from the Isabelle interactive theorem prover [33]. Our experimental results show that Vampire outperforms systems which could previously be used to solve such problems: higher-order theorem provers and satisfiability modulo theory (SMT) solvers.

The paper focuses on new, practical features extending first-order theorem provers for making them better suited for applications of reasoning in various theories, program analysis and verification. While the paper describes implementation details and challenges in the Vampire theorem prover, the described features and their implementation can be carried out in any other first-order prover.

Summarising, we believe that our paper advances the state-of-the-art in formal certification of programs and proofs. With the use of FOOL and polymorphic arrays, we bring first-order theorem proving closer to program logics and make first-order theorem proving better suited for program analysis and verification. We also believe that an implementation of FOOL advances automation of mathematics, making many problems using the boolean type directly understood by a first-order theorem prover, while they previously were treated as higher-order problems.

3.2 First Class Boolean Sort

Our recent work [28] presented a modification of many-sorted first-order logic that contains a boolean sort with a fixed interpretation and treats terms of the boolean sort as formulas. We called this logic FOOL, standing for first-order logic (FOL) + boolean sort. FOOL extends FOL by (i) treating boolean terms as formulas; (ii) `if-then-else` expressions; and (iii) `let-in` expressions. There is a model-preserving transformation of FOOL formulas to FOL formulas, hence an implementation of this transformation makes it possible to prove FOOL formulas using a first-order theorem prover.

The language of FOOL is, essentially, a superset of the core language of SMT-LIB 2 [7], the library of problems for SMT solvers. The difference between FOOL and the core language is that the former has richer `let-`

`in` expressions, which support local definitions of functions symbols of arbitrary arity, while the latter only supports local binding of variables.

FOOL can be regarded as the smallest superset of the SMT-LIB 2 Core language and TFF0. An implementation of a translation of FOOL to FOL thus also makes it possible to translate SMT-LIB problems to TPTP. Consider, for example, the following tautology, written in the SMT-LIB syntax: `(exists ((x Bool)) x)`. It quantifies over boolean variables and uses a boolean variable as a formula. Neither is allowed in the standard TPTP language, but can be directly expressed in an extended TPTP that represents FOOL.

The rest of this section presents features of FOOL not included in FOL, explains how they are implemented in Vampire and how they can be represented in an extended TPTP syntax understood by Vampire.

3.2.1 Proving with the Boolean Sort

Vampire supports many-sorted predicate logic and the TFF0 syntax for this logic. In many-sorted predicate logic all sorts are uninterpreted, while the boolean sort should be interpreted as a two-element set. There are several ways to support the boolean sort in a first-order theorem prover, for example, one can axiomatise it by adding two constants *true* and *false* of this sort and two axioms: $(\forall x : \mathit{bool})(x \doteq \mathit{true} \vee x \doteq \mathit{false})$ and $\mathit{true} \not\doteq \mathit{false}$. However, as we discuss in [28], using this axiomatisation in a superposition theorem prover may result in performance problems caused by self-paramodulation of $x \doteq \mathit{true} \vee x \doteq \mathit{false}$.

To overcome this problem, in [28] we proposed the following modification of the superposition calculus.

1. Use a special simplification ordering that makes the constants *true* and *false* smallest terms of the sort *bool* and also makes *true* greater than *false*.

2. Add the axiom $\mathit{true} \not\doteq \mathit{false}$.

3. Add a special inference rule, called *FOOL paramodulation*, of the form
$$\frac{C[s]}{C[\mathit{true}] \vee s \doteq \mathit{false}} \ ,$$
where

   (a) s is a term of the sort *bool* other than *true* and *false*;

   (b) s is not a variable;

Both ways of dealing with the boolean sort are supported in Vampire. The option `--fool_paramodulation`, which can be set to `on` or `off`, chooses one of them. The default value is `on`, which enables the modification.

Vampire uses the TFF0 subset of the TPTP syntax, which does not fully support FOOL. To write FOOL formulas in the input, one uses the standard TPTP notation: `$o` for the boolean sort, `$true` for *true* and `$false` for *false*. There are, however, two ways to output the boolean sort and the constants. One way will use the same notation as in the input and is the default, which is sufficient for most applications. The other way can be activated by the option `--show_fool on`, it will

1. denote as `$bool` every occurrence of *bool* as a sort of a variable or an argument (to a function or a predicate symbol);

2. denote as `$$true` every occurrence of *true* as an argument; and

3. denote as `$$false` every occurrence of *false* as an argument.

Note that an occurrence of any of the symbols `$bool`, `$$true` or `$$false` anywhere in an input problem is not recognised as syntactically correct by Vampire.

Setting `--show_fool` to `on` might be necessary if Vampire is used as a front-end to other reasoning tools. For example, one can use Vampire not only for proving, but also for preprocessing the input problem or converting it to clausal normal form. To do so, one uses the options `--mode preprocess` and `--mode clausify`, respectively. The output of Vampire can then be passed to other theorem provers, that either only deal with clauses or do not have sophisticated preprocessing. Setting `--show_fool` to `on` appends a definition of a sort denoted by `$bool` and constants denoted by `$$true` and `$$false` of this sort to the output. That way the output will always contain syntactically correct TFF0 formulas, which might not be true if the option is set to `off` (the default value).

Every formula of the standard FOL is syntactically a FOOL formula and has the same models. Vampire does not reason in FOOL natively, but rather translates the input FOOL formulas into FOL formulas in a way that preserves models. This is done at the first stage of preprocessing of the input problem.

Vampire implements the translation of FOOL formulas to FOL given in [28]. It involves replacing parts of the problem that are not syntactically correct in the standard FOL by applications of fresh function and predicate symbols. The set of assumptions is then extended by formulas

that define these symbols. Individual steps of the translation are displayed when the `--show_preprocessing` option is set to `on`.

In the next subsections we present the features of FOOL that are not present in FOL together with their syntax in the extended TFF0 and their implementation in Vampire.

3.2.2 Quantifiers over the Boolean Sort

FOOL allows quantification over *bool* and usage of boolean variables as formulas. For example, the formula $(\forall x : bool)(x \vee \neg x)$ is a syntactically correct tautology in FOOL. It is not however syntactically correct in the standard FOL where variables can only occur as arguments.

Vampire translates boolean variables to FOL in the following way. First, every formula of the form $x \Leftrightarrow y$, where x and y are boolean variables, is replaced by $x \doteq y$. Then, every occurrence of a boolean variable x anywhere other than in an argument is replaced by $x \doteq true$. For example, the tautology $(\forall x : bool)(x \vee \neg x)$ will be converted to the FOL formula $(\forall x : bool)(x \doteq true \vee x \not\doteq true)$ during preprocessing.

Note that it is possible to directly express quantified boolean formulas (QBF) in FOOL, and use Vampire to reason about them.

TFF0 does not support quantification over booleans. Vampire supports an extended version of TFF0 where the sort symbol `$o` is allowed to occur as the sort of a quantifier and boolean variables are allowed to occur as formulas. The formula $(\forall x : bool)(x \vee \neg x)$ can be expressed in this syntax as `![X:$o]: (X | ~X)`.

3.2.3 Functions and Predicates with Boolean Arguments

Functions and predicates in FOOL are allowed to take booleans as arguments. For example, one can define the logical implication as a binary function *impl* of the type $bool \times bool \to bool$ using the following axiom:

$$(\forall x : bool)(\forall y : bool)(impl(x, y) \Leftrightarrow \neg x \vee y).$$

Since Vampire supports many-sorted logic, this feature requires no additional implementation, apart from changes in the parser.

In TFF0, functions and predicates cannot have arguments of the sort `$o`. In the version of TFF0, supported by Vampire, this restriction is removed. Thus, the definition of *impl* can be expressed in the following way.

```
tff(impl, type, impl: ($o * $o) > $o).
tff(impl_definition, axiom,
    ![X:$o, Y:$o]: (impl(X, Y) <=> (~X | Y))).
```

3.2.4 Formulas as Arguments

Unlike the standard FOL, FOOL does not make a distinction between formulas and boolean terms. It means that a function or a predicate can take a formula as a boolean argument, and formulas can be used as arguments to equality between booleans. For example, with the definition of *impl*, given earlier, we can express in FOOL that P is a graph of a (partial) function of the type $\sigma \to \tau$ as follows:

$$(\forall x : \sigma)(\forall y : \tau)(\forall z : \tau) impl(P(x, y) \wedge P(x, z), y \doteq z). \tag{3.1}$$

Note that the definition of *impl* could as well use equality instead of equivalence.

In order to support formulas occurring as arguments, Vampire does the following. First, every expression of the form $\varphi \doteq \psi$ is replaced by $\varphi \Leftrightarrow \psi$. Then, for each formula ψ occurring as an argument the following translation is applied. If ψ is a nullary predicate $\top$ or $\bot$, it is replaced by *true* or *false*, respectively. If ψ is a boolean variable, it is left as is. Otherwise, the translation is done in several steps. Let $x_1, \ldots, x_n$ be all free variables of ψ and $\sigma_1, \ldots, \sigma_n$ be their sorts. Then Vampire

1. introduces a fresh function symbol g of the type

$$\sigma_1 \times \ldots \times \sigma_n \to bool;$$

2. adds the definition

$$(\forall x_1 : \sigma_1) \ldots (\forall x_n : \sigma_n)(\psi \Leftrightarrow g(x_1, \ldots, x_n) \doteq true)$$

   to its set of assumptions;

3. replaces ψ by $g(x_1, \ldots, x_n)$.

For example, after this translation has been applied for both arguments of *impl*, (3.1) becomes

$$(\forall x : \sigma)(\forall y : \sigma)(\forall z : \sigma) impl(g_1(x, y, z), g_2(y, z)),$$

where g_1 and g_2 are fresh function symbol of the types $\sigma \times \tau \times \tau \to bool$ and $\tau \times \tau \to bool$, respectively, defined by the following formulas:

1. $(\forall x : \sigma)(\forall y : \tau)(\forall z : \tau)(P(x, y) \wedge P(x, z) \Leftrightarrow g_1(x, y, z) \doteq \mathit{true})$;

2. $(\forall y : \tau)(\forall z : \tau)(y \doteq z \Leftrightarrow g_2(y, z) \doteq \mathit{true})$.

TFF0 does not allow formulas to occur as arguments. The extended version of TFF0, supported by Vampire, removes this restriction for arguments of the boolean sort. Formula (3.1) can be expressed in this syntax as follows:

```
![X:s, Y:t, Z:t]: impl(p(X, Y) & p(X, Z), Y = Z)
```

For a more interesting example, consider the following logical puzzle taken from the TPTP problem PUZ081:

> A very special island is inhabited only by knights and knaves. Knights always tell the truth, and knaves always lie. You meet two inhabitants: Zoey and Mel. Zoey tells you that Mel is a knave. Mel says, 'Neither Zoey nor I are knaves'. Who is a knight and who is a knave?

To solve the puzzle, one can formalise it as a problem in FOOL and give a corresponding extended TFF0 representation to Vampire. Let *zoye* and *mel* be terms of a fixed sort *person* that represent Zoye and Mel, respectively. Let *Says* be a predicate that takes a term of the sort *person* and a boolean term. We will write $\mathit{Says}(p, s)$ to denote that a person p made a logical statement s. Let *Knight* and *Knave* be predicates that take a term of the sort *person*. We will write $\mathit{Knight}(p)$ or $\mathit{Knave}(p)$ to denote that a person p is a knight or a knave, respectively. We will express the fact that knights only tell the truth and knaves only lie by axioms $(\forall p : \mathit{person})(\forall s : \mathit{bool})(\mathit{Knight}(p) \wedge \mathit{Says}(p, s) \Rightarrow s)$ and $(\forall p : \mathit{person})(\forall s : \mathit{bool})(\mathit{Knave}(p) \wedge \mathit{Says}(p, s) \Rightarrow \neg s)$, respectively. We will express the fact that every person is either a knight or a knave by the axiom $(\forall p : \mathit{person})(\mathit{Knight}(p) \oplus \mathit{Knave}(p))$, where $\oplus$ is the "exclusive or" connective. Finally, we will express the statements that Zoye and Mel make in the puzzle by axioms $\mathit{Says}(\mathit{zoye}, \mathit{Knave}(\mathit{mel}))$ and $\mathit{Says}(\mathit{mel}, \neg\mathit{Knave}(\mathit{zoye}) \wedge \neg\mathit{Knave}(\mathit{mel}))$, respectively.

The axioms and definitions, given above, can be written in the extended TFF0 syntax in the following way.

```
tff(person, type, person: $tType).
tff(says, type, says: (person * $o) > $o).

tff(knight, type, knight: person > $o).
tff(knights_always_tell_truth, axiom,
```

```
    ![P:person, S:$o]:
      (knight(P) & says(P, S) => S)).

tff(knave, type, knave: person > $o).
tff(knaves_always_lie, axiom,
    ![P:person, S:$o]:
      (knave(P) & says(P, S) => ~S)).

tff(very_special_island, axiom,
    ![P:person]: (knight(P) <~> knave(P))).

tff(zoey, type, zoey: person).
tff(mel, type, mel: person).

tff(zoye_says, hypothesis,
    says(zoey, knave(mel))).

tff(mel_says, hypothesis,
    says(mel, ~knave(zoey) & ~knave(mel))).
```

Vampire accepts this code, finds that the problem is satisfiable and outputs the saturated set of clauses. There one can see that Zoey is a knight and Mel is a knave. Note that the existing formalisations of this puzzle in TPTP (files `PUZ081^1.p`, `PUZ081^2.p` and `PUZ081^3.p`) employ the language of higher-order logic (THF) [44]. However, as we have just shown, one does not need to resort to reasoning in higher-order logic for this problem, and can enjoy the efficiency of reasoning in first-order logic.

This example makes one think about representing sentences in various epistemic or first-order modal logics in FOOL.

3.2.5 `if-then-else`

FOOL contains expressions of the form `if` ψ `then` s `else` t, where ψ is a boolean term, and s and t are terms of the same sort. The semantics of such expressions mirrors the semantics of conditional expressions in programming languages.

`if-then-else` expressions are convenient for expressing formulas coming from program analysis and interactive theorem provers. For example, consider the *max* function of the type $\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ that returns the maximum of its arguments. Its definition can be expressed in FOOL as

$$(\forall x : \mathbb{Z})(\forall y : \mathbb{Z})(max(x, y) \doteq \texttt{if } x \geq y \texttt{ then } x \texttt{ else } y). \tag{3.2}$$

To handle such expressions, Vampire translates them to FOL. This translation is done in several steps. Let $x_1, \ldots, x_n$ be all free variables of ψ, s and t, and $\sigma_1, \ldots, \sigma_n$ be their sorts. Let τ be the sort of both s and t. The steps of translation depend on whether τ is *bool* or a different sort. If τ is not *bool*, Vampire

1. introduces a fresh function symbol g of the type

$$\sigma_1 \times \ldots \times \sigma_n \to \tau;$$

2. adds the definitions

$$(\forall x_1 : \sigma_1) \ldots (\forall x_n : \sigma_n)(\psi \Rightarrow g(x_1, \ldots, x_n) \doteq s),$$
$$(\forall x_1 : \sigma_1) \ldots (\forall x_n : \sigma_n)(\neg\psi \Rightarrow g(x_1, \ldots, x_n) \doteq t)$$

   to its set of assumptions;

3. replaces `if` ψ `then` s `else` t by $g(x_1, \ldots, x_n)$.

If τ is *bool*, the following is different in the steps of translation:

1. a fresh predicate symbol g of the type $\sigma_1 \times \ldots \times \sigma_n$ is introduced instead; and

2. the added definitions use equivalence instead of equality.

For example, after this translation (3.2) becomes

$$(\forall x : \mathbb{Z})(\forall y : \mathbb{Z})(max(x, y) \doteq g(x, y)),$$

where g is a fresh function symbol of the type $\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ defined by the following formulas:

1. $(\forall x : \mathbb{Z})(\forall y : \mathbb{Z})(x \geq y \Rightarrow g(x, y) \doteq x)$;

2. $(\forall x : \mathbb{Z})(\forall y : \mathbb{Z})(x \not\geq y \Rightarrow g(x, y) \doteq y)$.

TPTP has two different expressions for `if-then-else`: `$ite_t` for constructing terms and `$ite_f` for constructing formulas. `$ite_t` takes a formula and two terms of the same sort as arguments. `$ite_f` takes three formulas as arguments.

Since FOOL does not distinguish formulas and boolean terms, it does not require separate expressions for the formula-level and term-level `if-then-else`. The extended version of TFF0, supported by Vampire, uses a new expression `$ite`, that unifies `$ite_t` and `$ite_f`. `$ite` takes a

formula and two terms of the same sort as arguments. If the second and the third arguments are boolean, such `$ite` expression is equivalent to `$ite_f`, otherwise it is equivalent to `$ite_t`.

Consider, for example, the above definition of *max*. It can be encoded in the extended TFF0 as follows.

```
tff(max, type, max: ($int * $int) > $int).
tff(max_definition, axiom,
    ![X:$int, Y:$int]:
      (max(X, Y) = $ite($greatereq(X, Y), X, Y))).
```

It uses the TPTP notation `$int` for the sort of integers and `$greatereq` for the greater-than-or-equal-to comparison of two numbers.

Consider now the following valid property of *max*:

$$(\forall x : \mathbb{Z})(\forall y : \mathbb{Z})(\texttt{if } max(x, y) \doteq x \texttt{ then } x \geq y \texttt{ else } y \geq x). \tag{3.3}$$

Its encoding in the extended TFF0 can use the same `$ite` expression:

```
![X:$int, Y:$int]: $ite(max(X, Y) = X,
                        $greatereq(X, Y),
                        $greatereq(Y, X)).
```

Note that TFF0 without `$ite` has to differentiate between terms and formulas, and so requires to use `$ite_t` in (3.2) and `$ite_f` in (3.3).

3.2.6 `let-in`

FOOL contains `let-in` expressions that can be used to introduce local function definitions. They have the form

$$\begin{aligned} &\texttt{let } f_1(x_1^1 : \sigma_1^1, \ldots, x_{n_1}^1 : \sigma_{n_1}^1) \;=\; s_1; \\ &\qquad \ldots \\ &\qquad f_m(x_1^m : \sigma_1^m, \ldots, x_{n_m}^m : \sigma_{n_m}^m) \;=\; s_m \\ &\texttt{ in } t, \end{aligned} \tag{3.4}$$

where

1. $m \geq 1$;
2. $f_1, \ldots, f_m$ are pairwise distinct function symbols;
3. $n_i \geq 0$ for each $1 \leq i \leq m$;
4. $x_1^i \ldots, x_{n_i}^i$ are pairwise distinct variables for each $1 \leq i \leq m$; and

5. $s_1, \ldots, s_m$ and t are terms.

The semantics of `let-in` expressions in FOOL mirrors the semantics of simultaneous non-recursive local definitions in programming languages. That is, $s_1, \ldots, s_m$ do not use the bindings of $f_1, \ldots, f_m$ created by this definition.

Note that an expression of the form (3.4) is not in general equivalent to m nested `let-ins`

$$\begin{array}{l} \texttt{let } f_1(x_1^1 : \sigma_1^1, \ldots, x_{n_1}^1 : \sigma_{n_1}^1) \;=\; s_1 \texttt{ in} \\ \quad \ddots \\ \quad\quad \texttt{let } f_m(x_1^m : \sigma_1^m, \ldots, x_{n_m}^m : \sigma_{n_m}^m) \;=\; s_m \texttt{ in} \\ \quad\quad\quad t. \end{array} \tag{3.5}$$

The main application of `let-in` expressions is in problems coming from program analysis, namely modelling of assignments. Consider for example the following code snippet featuring operations over an integer `array`.

```
array[3] := 5;
array[2] + array[3];
```

It can be translated to FOOL in the following way. We represent the integer array as an uninterpreted function *array* of the type $\mathbb{Z} \to \mathbb{Z}$ that maps an index to the array element at that index. The assignment of an array element can be translated to a combination of `let-in` and `if-then-else`.

$$\begin{array}{l} \texttt{let } \mathit{array}(i : \mathbb{Z}) \;=\; \texttt{if } i \doteq 3 \texttt{ then } 5 \texttt{ else } \mathit{array}(i) \texttt{ in} \\ \quad \mathit{array}(2) + \mathit{array}(3) \end{array} \tag{3.6}$$

Multiple bindings in a `let-in` expression can be used to concisely express simultaneous assignments that otherwise would require renaming. In the following example, constants a and b are swapped by a `let-in` expression. The resulting formula is equivalent to $f(b, a)$.

$$\texttt{let } a \;=\; b;\; b \;=\; a \texttt{ in } f(a, b) \tag{3.7}$$

In order to handle `let-in` expressions Vampire translates them to FOL. This is done in three stages for each expression in (3.4).

1. For each function symbol f_i where $0 \leq i < m$ that occurs freely in any of $s_{i+1}, \ldots, s_m$, introduce a fresh function symbol g_i. Replace all free occurrences of f_i in t by g_i.

2. Replace the `let-in` expression by an equivalent one of the form (3.5). This is possible because the necessary condition was satisfied by the previous step.

3. Apply a translation to each of the `let-in` expression with a single binding, starting with the innermost one.

The translation of an expression of the form

$$\texttt{let}\ f(x_1 : \sigma_1, \ldots, x_n : \sigma_n) = s\ \texttt{in}\ t$$

is done by the following sequence of steps. Let $y_1, \ldots, y_m$ be all free variables of s and t, and $\tau_1, \ldots, \tau_m$ be their sorts. Note that the variables in $x_1, \ldots, x_n$ are not necessarily disjoint from the variables in $y_1, \ldots, y_m$. Let σ_0 be the sort of s. The steps of translation depend on whether σ_0 is *bool* and not. If σ_0 is not *bool*, Vampire

1. introduces a fresh function symbol g of the type

$$\sigma_1 \times \ldots \times \sigma_n \times \tau_1 \times \ldots \times \tau_m \to \sigma_0;$$

2. adds to the set of assumptions the definition

$$\begin{aligned}&(\forall z_1 : \sigma_1) \ldots (\forall z_n : \sigma_n)(\forall y_1 : \tau_1) \ldots (\forall y_m : \tau_m)\\ &\quad (g(z_1, \ldots, z_n, y_1, \ldots, y_m) \doteq s'),\end{aligned}$$

   where $z_1, \ldots, z_n$ is a fresh sequence of variables and s' is obtained from s by replacing all free occurrences of $x_1, \ldots, x_n$ by $z_1, \ldots, z_n$, respectively; and

3. replaces $\texttt{let}\ f(x_1 : \sigma_1, \ldots, x_n : \sigma_n) = s\ \texttt{in}\ t$ by t', where t' is obtained from t by replacing all bound occurrences of $y_1, \ldots, y_m$ by fresh variables and each application $f(t_1, \ldots, t_n)$ of a free occurrence of f by $g(t_1, \ldots, t_n, y_1, \ldots, y_m)$.

If σ_0 is *bool*, the steps of translation are different:

1. a fresh predicate symbol of the type

$$\sigma_1 \times \ldots \times \sigma_n \times \tau_1 \times \ldots \times \tau_m$$

   is introduced instead;

2. the added definition uses equivalence instead of equality.

For example, after this translation (3.6) becomes $g(2) + g(3)$, where g is a fresh function symbol of the type $\mathbb{Z} \to \mathbb{Z}$ defined by the following formula:

$$(\forall i : \mathbb{Z})(g(i) \doteq \texttt{if}\ i \doteq 3\ \texttt{then}\ 5\ \texttt{else}\ \mathit{array}(i)).$$

The example (3.7) is translated in the following way. First, the `let-in` expression is translated to the form (3.5). The constant a has a free occurrence in the body of b, therefore it is replaced by a fresh constant a'. The formula (3.7) becomes

$$\begin{array}{l} \texttt{let}\ a' = b\ \texttt{in} \\ \quad \texttt{let}\ b = a\ \texttt{in} \\ \quad\quad f(a', b). \end{array}$$

Then, the translation is applied to both `let-in` expressions with a single binding and the resulting formula becomes $f(a'', b')$, where a'' and b' are fresh constants, defined by formulas $a'' \doteq b$ and $b' \doteq a$.

TPTP has four different expressions for `let-in`: `$let_tt` and `$let_ft` for constructing terms, and `$let_tf` and `$let_ff` for constructing formulas. All of them denote a single binding. `$let_tt` and `$let_tf` denote a binding of a function symbol, whereas `$let_ft` and `$let_ff` denote a binding of a predicate symbol. All four expressions take a (possibly universally quantified) equation as the first argument and a term (in case of `$let_tt` and `$let_ft`) or a formula (in case of `$let_tf` and `$let_ff`) as the second argument. TPTP does not provide any notation for `let-in` expressions with multiple bindings.

Similarly to `if-then-else`, `let-in` expressions in FOOL do not need different notation for terms and formulas. The modification of TFF0 supported by Vampire introduces a new `$let` expression, that unifies `$let_tt`, `$let_ft`, `$let_tf` and `$let_ff`, and extends them to support multiple bindings. Depending on whether the binding is of a function or predicate symbol and whether the second argument of the expression is term or formula, a `$let` expression is equivalent to one of `$let_tt`, `$let_ft`, `$let_tf` and `$let_ff`.

The new `$let` expressions use different syntax for bindings. Instead of a quantified equation, they use the following syntax: a function symbol possibly followed by a list of variable arguments in parenthesis, followed by the `:=` operator and the body of the binding. Similarly to quantified variables, variable arguments are separated with commas and each variable might include a sort declaration. A sort declaration can be omitted, in which case the variable is assumed to the be of the sort of individuals

($i).

Formula (3.6) can be written in the extended TFF0 with the TPTP interpreted function $sum, representing integer addition, as follows:

```
$let(array(I:$int) := $ite(I = 3, 5, array(I)),
     $sum(array(2), array(3))).
```

The same $let expression can be used for multiple bindings. For that, the bindings should be separated by a semicolon and passed as the first argument. The formula (3.7) can be written using $let as follows.

```
$let(a := b; b := a, f(a, b)))
```

Overall, $ite and $let expressions provide a more concise syntax for TPTP formulas than the TFF0 variations of if-then-else and let-in expressions. To illustrate this point, consider the following snippet of TPTP code, taken from the TPTP problem SYN000_2.

```
tff(let_binders, axiom, ![X:$i]:
    $let_ff(![Y1:$i, Y2:$i]: (q(Y1, Y2) <=> p(Y1)),
      q($let_tt(![Z1:$i]:
          (f(Z1) = g(Z1, b)), f(a)), X) &
      p($let_ft(![Y3:$i, Y4:$i]: (q(Y3, Y4) <=>
          $ite_f(Y3 = Y4, q(a, a), q(Y3, Y4))),
          $ite_t(q(b, b), f(a), f(X)))))).
```

It uses both of the TFF0 variations of if-then-else and three different variations of let-in. The same snippet can be expressed more concisely using $ite and $let expressions.

```
tff(let_binders, axiom, ![X:$i]:
    $let(q(Y1, Y2) := p(Y1),
      q($let(f(Z1) := g(Z1, b), f(a)), X) &
      p($let(q(Y3, Y4) := $ite(Y3 = Y4, q(a, a), q(Y3,Y4))),
          $ite(q(b, b), f(a), f(X)))))).
```

3.3 Polymorphic Theory of Arrays

Using built-in arrays and reasoning in the first-order theory of arrays are common in program analysis, for example for finding loop invariants in programs using arrays [29]. Previous versions of Vampire supported theories of integer arrays and arrays of integer arrays [30]. No other array sorts were supported and in order to implement one it would be necessary to hardcode a new sort and add the theory axioms corresponding to

that sort. In this section we describe a polymorphic theory of arrays implemented in Vampire.

3.3.1 Definition

The polymorphic theory of arrays is the union of theories of arrays parametrised by two sorts: sort τ of indexes and sort σ of values. It would have been proper to call these theories the theories of maps from τ to σ, however we decided to call them arrays for the sake of compatibility with arrays as defined in SMT-LIB.

A theory of arrays is a first-order theory that contains a sort $array(\tau, \sigma)$, function symbols $select : array(\tau, \sigma) \times \tau \to \sigma$ and $store : array(\tau, \sigma) \times \tau \times \sigma \to array(\tau, \sigma)$, and three axioms. The function symbol *select* represents a binary operation of extracting an array element by its index. The function symbol *store* represents a ternary operation of updating an array at a given index with a given value. The array axioms are:

1. read-over-write 1

$$(\forall a : array(\tau, \sigma))(\forall v : \sigma)(\forall i : \tau)(\forall j : \tau) \\ (i \doteq j \Rightarrow select(store(a, i, v), j) \doteq v);$$

2. read-over-write 2

$$(\forall a : array(\tau, \sigma))(\forall v : \sigma)(\forall i : \tau)(\forall j : \tau) \\ (i \not\doteq j \Rightarrow select(store(a, i, v), j) \doteq select(a, j));$$

3. extensionality

$$(\forall a : array(\tau, \sigma))(\forall b : array(\tau, \sigma)) \\ ((\forall i : \tau)(select(a, i) \doteq select(b, i)) \Rightarrow a \doteq b).$$

We will call every concrete instance of the theory of arrays for concrete sorts τ and σ the (τ, σ)-*instance*.

One can use the polymorphic theory of arrays to express program properties. Recall the code snippet involving arrays mentioned in Section 3.2:

```
array[3] := 5;
array[2] + array[3];
```

Formula (3.6) used an interpreted function to represent the array in this code. We can alternatively use arrays to represent it as follows

$$\begin{array}{l}\texttt{let } array \;=\; store(array, 3, 5) \texttt{ in} \\ \quad select(array, 2) + select(array, 3)\end{array} \tag{3.8}$$

3.3.2 Implementation in Vampire

Vampire implements reasoning in the polymorphic theory of arrays by adding corresponding sorts axioms when the input uses array sorts and/or functions.

Whenever the input problem uses a sort $array(\tau, \sigma)$, Vampire adds this sort and function symbols *select* and *store* of the types $array(\tau, \sigma) \times \tau \to \sigma$ and $array(\tau, \sigma) \times \tau \times \sigma \to array(\tau, \sigma)$, respectively.

If the input problem contains *store*, Vampire adds the following axioms for the sorts τ and σ used in the corresponding array theory instance:

$$\begin{array}{c}(\forall a : array(\tau, \sigma))(\forall i : \tau)(\forall v : \sigma) \\ (select(store(a, i, v), i) \doteq v)\end{array} \tag{3.9}$$

$$\begin{array}{l}(\forall a : array(\tau, \sigma))(\forall i : \tau)(\forall j : \tau)(\forall v : \sigma) \\ \quad (i \neq j \Rightarrow select(store(a, i, v), j) \doteq select(a, j))\end{array} \tag{3.10}$$

$$\begin{array}{l}(\forall a : array(\tau, \sigma))(\forall b : array(\tau, \sigma)) \\ \quad (a \neq b \Rightarrow (\exists i : \tau)(select(a, i) \not\doteq select(b, i)))\end{array} \tag{3.11}$$

These axioms are equivalent to the axioms read-over-write 1, read-over-write 2 and extensionality.

If the input contains only *select* but not *store* for this instance, then only extensionality (3.11) is added.

Theory axioms are not added when the `--theory_axioms` option is set to `off` (the default value is `on`), which leaves an option for the user to try her or his own axiomatisation of arrays.

Vampire uses the extensionality resolution rule [22] to efficiently reason with the extensionality axiom.

To express arrays, the TPTP syntax extension supported by Vampire allows, for every pair of sorts τ and σ, denoted by `t` and `s` in the TFF0 syntax, to denote the sort $array(\tau, \sigma)$ by `$array(s,t)`. Function symbols *select* and *store* can be expressed as ad-hoc polymorphic `$select` and `$store`, respectively for every pairs of sorts τ, σ. Previously, the theories of integer arrays and arrays of integer arrays were represented as sorts `$array1` and `$array2` in Vampire, with the corresponding sort-specific

function symbols `$select1`, `$select2`, `$store1` and `$store2`. Our new implementation in Vampire, with support for the polymorphic theory of arrays, deprecates these two concrete array theories. Instead, one can now use the sorts `$array($int,$int)` and `$array($int,$array($int, $int))`. For example, formula (3.8) can be written in the extended TFF0 syntax as follows:

```
$let(array := $store(array,3,5),
     $sum($select(array,2), $select(array,3))).
```

3.3.3 Theory of Boolean Arrays

An interesting special case of the polymorphic theory of arrays is the theory of boolean arrays. In that theory the *select* function has the type $array(\tau, bool) \times \tau \rightarrow bool$ and the *store* function has the type $array(\tau, bool) \times \tau \times bool \rightarrow array(\tau, bool)$. This means that applications of *select* can be used as formulas and *store* can have a formula as the third argument.

Vampire implements the theory of booleans arrays similarly to other sorts, by adding theory axioms when the option `--theory_axioms` is enabled. However, the theory axioms are different for the following reason. The axioms of the theory of boolean arrays are syntactically correct in FOOL but not in FOL, because they use quantification over booleans. However, Vampire adds theory axioms only after a translation of FOOL to FOL. For this reason, Vampire uses the following set of axioms for boolean arrays:

$$(\forall a : array(\tau, bool))(\forall i : \tau)(\forall v : bool) \\ (select(store(a, i, v), i) \Leftrightarrow (v \doteq true))$$

$$(\forall a : array(\tau, bool))(\forall i : \tau)(\forall j : \tau)(\forall v : bool) \\ (i \neq j \Rightarrow select(store(a, i, v), j) \Leftrightarrow select(a, j))$$

$$(\forall a : array(\tau, bool))(\forall b : array(\tau, bool)) \\ (a \doteq b \Rightarrow (\exists i : \tau)(select(a, i) \oplus select(b, i)))$$

where $\oplus$ is the "exclusive or" connective.

One can use the theory of boolean arrays, for example, to express properties of bit vectors. In the following example we give a formalisation of a basic property of XOR encryption, where the key, the message and the cipher are bit vectors. Let *encrypt* be a function of the type $array(\mathbb{Z}, bool) \times array(\mathbb{Z}, bool) \rightarrow array(\mathbb{Z}, bool)$. We will write

$encrypt(message, key)$ to denote the result of bit-wise application of the XOR operation to *message* and *key*. For simplicity we will assume that the message and the key are of equal length. The definition of *encrypt* can be expressed with the following axiom:

$$(\forall message : array(\mathbb{Z}, bool))(\forall key : array(\mathbb{Z}, bool))(\forall i : \mathbb{Z})$$
$$(select(encrypt(message, key), i) \doteq$$
$$select(message, i) \oplus select(key, i)).$$

An important property of XOR encryption is its vulnerability to the known plaintext attack. It means that knowing a message and its cipher, one can obtain the key that was used to encrypt the message by encrypting the message with the cipher. This property can be expressed by the following formula.

$$(\forall plaintext : array(\mathbb{Z}, bool))(\forall cipher : array(\mathbb{Z}, bool))$$
$$(\forall key : array(\mathbb{Z}, bool))(cipher \doteq encrypt(plaintext, key) \Rightarrow$$
$$key \doteq encrypt(plaintext, cipher))$$

The sort $array(\mathbb{Z}, bool)$ is represented in the extended TFF0 syntax as `$array($int,$bool)`. The presented property of XOR encryption can be expressed in the extended TFF0 in the following way.

```
tff(encrypt, type, encrypt: ($array($int,$o) *
    $array($int,$o)) > $array($int,$o)).

tff(xor_encryption, axiom,
    ![Message:$array($int,$o),
      Key:$array($int,$o), I:$int]:
      ($select(encrypt(Message, Key), I) =
        ($select(Message, I) <~> $select(Key,I)))).

tff(known_plaintext_attack, conjecture,
    ![Plaintext:$array($int,$o),
      Cipher:$array($int,$o), Key:$array($int,$o)]:
        ((Cipher = encrypt(Plaintext, Key)) =>
          (Key = encrypt(Plaintext, Cipher)))).
```

3.4 Program Analysis with the New Extensions

In this section we illustrate how FOOL makes first-order theorem provers better suited to applications in program analysis and verification. Firstly, we give concrete examples exemplifying the use of FOOL for expressing program properties. We avoid various program analysis steps, such as SSA form computations and renaming program variables; instead we show how program properties can directly be expressed in FOOL. We also present a technique for automatically generating the next state relation of any program with assignments, `if-then-else`, and sequential composition. For doing so, we introduce a simple extension of FOOL, allowing for a general translation that is linear in the size of the program. This is a new result intended to understand which extensions of first-order logic are adequate for naturally representing fragments of imperative programs.

3.4.1 Encoding the Next State Relation

Consider the program given in Figure 3.1, written in a C-like syntax, using a sequence of two conditional statements. The program first computes the maximal value *max* of two integers x and y and then adds the absolute value of *max* to x. A safety assertion, in FOL, is specified at the end of the loop, using the **assert** construct. This program is clearly safe, the assertion is satisfied. To prove program safety, one needs to reason about the program's transition relation, in particular reason about conditional statements, and express the final value of the program variable *res*. The partial correctness of the program of Figure 3.1 can be *automatically* expressed in FOOL, and then Vampire can be used to prove program safety. This requires us to encode (i) the next state value of *res* (and

```
res := x;
if (x > y)
    then max := x;
    else max := y;
if (max > 0)
    then res := res + max;
    else res := res − max;

assert res ≥ x
```

Figure 3.1. Sequence of conditionals.

```
tff(x, type, x: $int).
tff(y, type, y: $int).
tff(max, type, max: $int).
tff(res, type, res: $int).
tff(res1, type, res1: $int).

tff(transition_relation, hypothesis,
    res1 = $let(res := x,
            $let(max := $ite($greater(x, y),
                             $let(max := x, max),
                             $let(max := y, max)),
              $let(res := $ite($greater(max, 0),
                               $let(res := $sum(res, max), res),
                               $let(res := $difference(res, max),
                                    res)),
                res)))).

tff(safety_property, conjecture, $greatereq(res1, x)).
```

Figure 3.2. Representation of the partial correctness statement of the code on Figure 3.1 in Vampire.

max) as a hypothesis in the extended TFF0 syntax of FOOL, by using the `if-then-else` (`$ite`) and `let-in` (`$let`) constructs, and (ii) the safety property as the conjecture to be proven by Vampire.

Figure 3.2 shows this extended TFF0 encoding. The use of `if-then-else` and `let-in` constructs allows us to have a direct encoding of the transition relation of Figure 3.1 in FOOL. Note that each expression from the program appears only once in the encoding.

We now explain how the encoding of the next state values of program variables can be generated automatically. We consider programs using assignments `:=`, `if-then-else` and sequential composition ;. We begin by making an assumption about the structure of programs (which we relax later). A program P is in *restricted form* if for any subprogram of the form **if** e **then** P_1 **else** P_2 the subprograms P_1 and P_2 only make assignments to the same single variable. Given a program P in restricted form let us define its translation $[P]$ inductively as follows:

- [x := e] is let x = e in x;
- [**if** e **then** P_1 **else** P_2], where P_1 and P_2 update x, is let x = if e then $[P_1]$ else $[P_2]$ in x;
- $[P_1;P_2]$ is let D in $[P_2]$ where $[P_1]$ is let D in x.

```
if (x > y)
   then t := x; x := y; y := t;
assert y ≥ x
```

Figure 3.3. Updating multiple variables.

Given a program P, the next state value for variable x can be given by $[P;\ \mathtt{x}\ :=\ \mathtt{x}]$, i.e. by ensuring the final statement of the program updates the variable of interest. The restricted form is required as conditionals must be viewed as assignments in the translation and assignments can only be made to single variables.

To demonstrate the limitations of this restriction let us consider the simple program in Figure 3.3 that ensures that x is not larger than y. We cannot apply the translation as the conditional updates three variables. To generalise the approach we can extend FOOL with *tuple expressions*, let us call this extension FOOL+. In this extended logic the next state values for Figure 3.3 can be encoded as follows:

$$
\begin{array}{ll}
\texttt{let}\ (x,y,t) = & \texttt{if}\ x > y\ \texttt{then} \\
& \quad \texttt{let}\ (x,y,t) = (x,y,x)\ \texttt{in} \\
& \quad\quad \texttt{let}\ (x,y,t) = (y,y,t)\ \texttt{in} \\
& \quad\quad\quad \texttt{let}\ (x,y,t) = (x,t,t)\ \texttt{in}\ (x,y,t) \\
& \quad \texttt{else}\ (x,y,t) \\
\texttt{in}\ (x,y,t) &
\end{array}
$$

We now give a brief sketch of the extended logic FOOL+ and the associated translation. We omit details since its full definition and semantics would require essentially repeating definitions from [28]. FOOL+ extends FOOL by tuples; for all expressions t_i of type σ_i we can use a *tuple expression* $(t_1, \ldots, t_n)$ of type $(\sigma_1, \ldots, \sigma_n)$. The logic should also include a suitable tuple projection function, which we do not discuss here.

This extension allows for a more general translation in two senses: first, the previous restricted form is lifted; and second, it now gives the next state values of *all* variables updated by the program. Given a program P its translation $[P]$ will have the form $\texttt{let}\ (x_1, \ldots, x_n) = E\ \texttt{in}\ (x_1, \ldots, x_n)$, where $x_1, \ldots, x_n$ are all variables updated by P, that is, all variables used in the left-hand-side of an assignment. We inductively define $[P]$ as follows:

- $[\mathtt{x}_i\ :=\ e]$ is $\texttt{let}\ (\ldots, x_i, \ldots) = (\ldots, e, \ldots)\ \texttt{in}\ (x_1, \ldots, x_n)$,
- $[\underline{\textbf{if}}\ \mathrm{e}\ \underline{\textbf{then}}\ P_1\ \underline{\textbf{else}}\ P_2]$ is $\texttt{let}\ (x_1, \ldots, x_n) = \texttt{if}\ e\ \texttt{then}\ [P_1]\ \texttt{else}\ [P_2]$

`in` $(x_1, \ldots, x_n)$,

- $[P_1;P_2]$ is `let` D `in` $[P_2]$ where $[P_1]$ is `let` D `in` $(x_1, \ldots, x_n)$.

This translation is bounded by $O(v \cdot n)$, where v is the number of variables in the program and n is the program size (number of statements) as each program statement is used once with one or two instances of $(x_1, \ldots, x_n)$. This becomes $O(n)$ if we assume that the number of variables is fixed. The translation could be refined so that some introduced `let-in` expressions only use a subset of program variables. Finally, this translation preserves the semantics of the program.

Theorem 5. Let P be a program with variables $(x_1, \ldots, x_n)$ and let $u_1, \ldots, u_n, v_1, \ldots, v_n$ be values (where u_i and v_i are of the same type as x_i). If P changes the state $\{x_1 \to u_1, \ldots, x_n \to u_n\}$ to $\{x_1 \to v_1, \ldots, x_n \to v_n\}$ then the value of $[P]$ in $\{x_1 \to u_1, \ldots, x_n \to u_n\}$ is $(v_1, \ldots, v_n)$.

This translation encodes the next state values of program variables by directly following the structure of the program. This leads to a succinct representation that, importantly, does not lose any information or attempt to translate the program too early. This allows the theorem prover to apply its own translation to FOL that it can handle efficiently. While FOOL+ is not yet fully supported in Vampire, we believe experimenting with FOOL+ on examples coming from program analysis and verification is an interesting task for future work.

3.4.2 A Program with a Loop and Arrays

Let us now show the use of FOOL in Vampire for reasoning about programs with loops. Consider the program given in Figure 3.4, written in a C-like syntax. The program fills an integer-valued array B by the strictly positive values of a source array A, and an integer-valued array C with the non-positive values of A. A safety assertion, in FOL, is specified at the end of the loop, using the **assert** construct. The program of Figure 3.4 is clearly safe as the assertion is satisfied when the loop is exited. However, to prove program safety we need additional loop properties, that is loop invariants, that hold at any loop iteration. These can be automatically generated using existing approaches, for example the symbol elimination method for invariant generation in Vampire [29]. In this case we use the FOL property specified in the **invariant** construct of Figure 3.4. This invariant property states that at any loop iteration, (i) the sum of visited array elements in A is the sum of visited elements in B and C (that is, $a = b + c$), (ii) the number of visited array elements in A, B, C is positive

$a := 0; b := 0; c := 0;$

invariant $\mathrm{a} = \mathrm{b} + \mathrm{c} \wedge$
$\mathrm{a} \geq 0 \wedge \mathrm{b} \geq 0 \wedge \mathrm{c} \geq 0 \wedge \mathrm{a} \leq \mathrm{k} \wedge$
$(\forall p)(0 \leq p < b \Rightarrow (\exists i)(0 \leq i < a \wedge A[i] > 0 \wedge B[p] = A[i]))$

while $(a \leq k)$ **do**
  if $(A[a] > 0)$
    then $B[b] := A[a]; b := b + 1;$
    else $C[c] := A[a]; c := c + 1;$
  $a := a + 1;$
end do

assert $(\forall p)(0 \leq p < b \Rightarrow B[p] > 0)$

Figure 3.4. Array partition.

(that is, $a \geq 0$, $b \geq 0$, and $c \geq 0$), with $a \leq k$, and (iii) each array element $B[0], \ldots, B[b-1]$ is a strictly positive element in A. Formulating the latter property requires quantifier alternation in FOL, resulting in the quantified property with $\forall\exists$ listed in the invariant of Figure 3.4. We can verify the safety of the program using Hoare-style reasoning in Vampire. The partial correctness property is that the invariant and the negation of the loop condition implies the safety assertion. This is the conjecture to be proven by Vampire. Figure 3.5 shows the encoding in the extended TFF0 syntax of this partial correctness statement; note that this uses the built-in theory of polymorphic arrays in Vampire, where $arrayA$, $arrayB$ and $arrayC$ correspond respectively to the arrays A, B and C.

So far, we assumed that the given invariant in Figure 3.4 is indeed an invariant. Using FOOL+ described in Section 3.4.1, we can verify the inductiveness property of the invariant, as follows: (i) express the the transition relation of the loop in FOOL+, and (ii) prove that, if the invariant holds at an arbitrary loop iteration i, then it also holds at loop iteration $i + 1$. For proving this, we can again use FOOL+ to formulate the next state values of loop variables in the invariant at loop iteration $i + 1$. Moreover, FOOL+ can also be used to express formulas as inputs to the symbol elimination method for invariant generation in Vampire. We leave the task of using FOOL+ for invariant generation as further work.

```
tff(a, type, a: $int).
tff(b, type, b: $int).
tff(c, type, c: $int).
tff(k, type, k: $int).
tff(arrayA, type, arrayA: $array($int, $int)).
tff(arrayB, type, arrayB: $array($int, $int)).
tff(arrayC, type, arrayC: $array($int, $int)).

tff(invariant_property, hypothesis,
    inv <=> ((a = $sum(b, c)) & $greatereq(a, 0) & $greatereq(b, 0)
                              & $greatereq(c, 0) & $lesseq(a, k) &
            ![P:$int]: ($lesseq(0, P) & $less(P, b) =>
              (?[I:$int]: ($lesseq(0, I) & $less(I, a) &
                 $greater($select(arrayA, I), 0) &
                 $select(arrayB, P) = $select(arrayA, I))))))).

tff(safety_property, conjecture,
    (inv & ~$lesseq(a, k)) =>
      (![P:$int]: ($lesseq(0, P) & $less(P, b) =>
                   $greater($select(arrayB, P), 0)))).
```

Figure 3.5. Representation of the partial correctness statement of the code on Figure 3.4 in Vampire.

3.5 Experimental Results

The extension of Vampire to support FOOL and the polymorphic theory of arrays comprises about 3,100 lines of C++ code, of which the translation of FOOL to FOL and FOOL paramodulation takes about 2,000 lines, changes in the parser about 500 lines and the implementation of the polymorphic theory of arrays about 600 lines. Our implementation is available at `www.cse.chalmers.se/~evgenyk/fool-experiments/` and will be included in the forthcoming official release of Vampire.

In the sequel, by Vampire we mean its version including support for FOOL and the polymorphic theory of arrays. We write Vampire* for its version with FOOL paramodulation turned off.

In this section we present experimental results obtained by running Vampire on FOOL problems. Unfortunately, no large collections of such problems are available, because FOOL was not so far supported by any first-order theorem prover. What we did was to extract such benchmarks from other collections.

1. We noted that many problems in the higher-order part of the TPTP

library [42] are FOOL problems, containing no real higher-order features. We converted them to FOOL problems.

2. We used a collection of first-order problems about (co)algebraic datatypes, generated by the Isabelle theorem prover [33], see Subsection 3.5.2 for more details.

Our results are summarised in Tables 3.1–3.3 and discussed below. These results were obtained on a MacBook Pro with a 2,9 GHz Intel Core i5 and 8 Gb RAM, and using the time limit of 60 seconds per problem. Both the benchmarks and the results are available at `www.cse.chalmers.se/~evgenyk/fool-experiments/`.

3.5.1 Experiments with TPTP Problems

The higher-order part of the TPTP library contains 3036 problems. Among these problems, 134 contain either boolean arguments in function applications or quantification over booleans, but contain no lambda abstraction, higher-order sorts or higher-order equality. We used these 134 problems, since they belong to FOOL but not to FOL. We translated these problems from THF0 to the modification of TFF0, supported by Vampire using the following syntactic transformation: (a) every occurrence of the keyword `thf` was replaced by `tff`; (b) every occurrence of a sort definition of the form `s_1 > ... > s_n > s` was replaced by `s_1 * ... * s_n > s`; (c) every occurrence of a function application of the form `f @ t_1 @ ... @ t_n` was replaced by `f(t_1, ..., t_n)`.

Out of 134 problems, 123 were marked as Theorem and 5 as Unsatisfiable, 5 as CounterSatisfiable, and 1 as Satisfiable, using the SZS status of TPTP. Essentially, this means that among their satisfiability-checking analogues, 128 are unsatisfiable and 6 are satisfiable. Vampire was run with the `--mode casc` option for unsatisfiable (Theorem and Unsatisfiable) problems and with `--mode casc_sat` for satisfiable (CounterSatisfiable and Satisfiable) problems. These options correspond to the CASC competition modes of Vampire for respectively proving validity (i.e. unsatisfiability) and satisfiability of an input problem.

For this experiment, we compared the performance of Vampire with those of the higher-order theorem provers used in the the latest edition of CASC [43]: Satallax [14], Leo-II [9], and Isabelle [33]. We note that all of them used the first-order theorem prover E [39] for first-order reasoning (Isabelle also used several other provers).

Table 3.1 summarises our results on these problems. Only Vampire, Vampire* and Satallax were able to solve all of them, while Vampire was

Table 3.1. Runtimes in seconds of provers on the set of 134 higher-order TPTP problems.

Prover	Solved	Total time on solved problems
Vampire	134	3.59
Vampire⋆	134	7.28
Satallax	134	23.93
Leo-II	127	27.42
Isabelle	128	893.80

the fastest among all provers. We believe these results are significant for two reasons. First, for solving these problems previously one needed higher-order theorem provers, but now can they be proven using first-order reasoners. Moreover, even on such simple problems there is a clear gain from using FOOL paramodulation.

3.5.2 Experiments with Algebraic Datatypes Problems

For this experiment, we used 152 problems generated by the Isabelle theorem prover. These problems express various properties of (co)algebraic datatypes and are written in the SMT-LIB 2 syntax [7]. All 152 problems contain quantification over booleans, boolean arguments in function/predicate applications and `if-then-else` expressions. These examples were generated and given to us by Jasmin Blanchette, following the recent work on reasoning about (co)datatypes [36]. To run the benchmark we first translated the SMT-LIB files to the TPTP syntax using the SMTtoTPTP translator [8] version 0.9.2. Let us note that this version of SMTtoTPTP does not fully support the boolean type in SMT-LIB. However, by setting the option `--keepBool` in SMTtoTPTP, we managed to translate these 152 problems into an extension of TFF0, which Vampire can read. We also modified the source code of SMTtoTPTP so that `if-then-else` expressions in the SMT-LIB files are not expanded but translated to `$ite` in FOOL. A similar modification would have been needed for translating `let-in` expressions; however, none of our 152 examples used `let-in`.

After translating these 152 problems into an extended TFF0 syntax supporting FOOL, we ran Vampire twice on each benchmark: once using the option `--mode casc`, and once using `--mode casc_sat`. For each problem, we recorded the fastest successful run of Vampire. We used a similar setting for evaluating Vampire⋆. In this experiment, we then

Table 3.2. Runtimes in seconds of provers on the set of 152 algebraic datatypes problems.

Prover	Solved	Total time on solved problems
Vampire	59	26.580
Z3	57	4.291
Vampire ⋆	56	26.095
CVC4	53	25.480

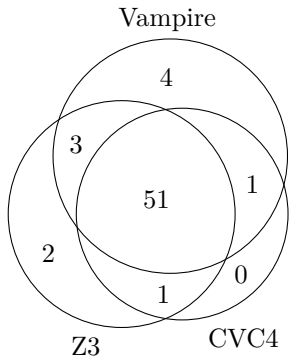

Figure 3.6. Venn diagram of the subsets of the algebraic datatypes problems, solved by Vampire, CVC4 and Z3.

compared Vampire with the best available SMT solvers, namely with CVC4 [6] and Z3 [20].

Table 3.2 summarises the results of our experiments on these 152 problems. Vampire solved the largest number of problems, and all problems solved by Vampire ⋆ were also solved by Vampire. Figure 3.6 shows the Venn diagram of the sets of problems solved by Vampire, CVC4 and Z3, where the numbers denote the numbers of solved problems. All problems apart from 11 were either solved by all systems or not solved by all systems. Table 3.3 details performance results on these 11 problems.

Based on our experimental results shown in Tables 3.2 and 3.3, we make the following observations. On the given set of problems the implementation of FOOL reasoning in Vampire was efficient enough to compete with state-of-the-art SMT solvers. This is significant because the problems were tailored for SMT reasoning. Vampire not only solved the largest number of problems, but also yielded runtime results that are comparable with those of CVC4. Whenever successful, Z3 turned out to be faster than Vampire; we believe this is because of the sophisticated

Table 3.3. Runtimes in seconds of provers on selected algebraic datatypes problems. Dashes mean the solver failed to find a solution.

Problem	Vampire	CVC4	Z3
afp/abstract_completeness/1830522	—	—	0.172
afp/bindag/2193162	—	—	0.388
afp/coinductive_stream/2123602	—	0.373	0.101
afp/coinductive_stream/2418361	3.392	—	—
afp/huffman/1811490	0.023	—	—
afp/huffman/1894268	0.025	—	0.052
distro/gram_lang/3158791	0.047	0.179	—
distro/koenig/1759255	0.070	—	—
distro/rbt_impl/1721121	4.523	—	—
distro/rbt_impl/2522528	0.853	—	0.064
gandl/bird_bnf/1920088	0.037	—	0.077

preprocessing steps in Z3. Improving FOOL preprocessing in Vampire, for example for more efficient CNF translation of FOOL formulas, is an interesting task for further research. We note that the usage of FOOL paramodulation showed improvement.

3.6 Related Work

FOOL was introduced in our previous work [28]. This also presented a translation from FOOL to the ordinary first-order logic, and FOOL paramodulation. In this paper we describe the first practical implementation of FOOL and FOOL paramodulation.

Superposition theorem proving in finite domains, such as the boolean domain, is also discussed in [24]. The approach of [24] sometimes falls back to enumerating instances of a clause by instantiating finite domain variables with all elements of the corresponding domains. Nevertheless, it allows one to also handle finite domains with more than two elements. One can also generalise our approach to arbitrary finite domains by using binary encodings of finite domains. However, this will necessarily result in loss of efficiency, since a single variable over a domain with 2^k elements will become k variables in our approach, and similarly for function arguments. Although [24] reports preliminary results with the theorem prover SPASS, we could not make an experimental comparison since the SPASS implementation has not yet been made public.

Handling boolean terms as formulas is common in the SMT community. The SMT-LIB project [7] defines its core logic as first-order logic extended with the distinguished first-class boolean sort and the `let-in` expression used for local bindings of variables. The language of FOOL extends the SMT-LIB core language with local function definitions, using `let-in` expressions defining functions of arbitrary, and not just zero, arity.

A recent work [8] presents SMTtoTPTP, a translator from SMT-LIB to TPTP. SMTtoTPTP does not fully support boolean sort, however one can use SMTtoTPTP with the `--keepBool` option to translate SMT-LIB problems to the extended TFF0 syntax, supported by Vampire.

Our implementation of the polymorphic theory of arrays uses a syntax that coincides with the TPTP's own syntax for polymorphically typed first-order logic TFF1 [11].

3.7 Conclusion and Future Work

We presented new features recently implemented in Vampire. They include FOOL: the extension of first-order logic by a first-class boolean sort, `if-then-else` and `let-in` expressions, and polymorphic arrays. Vampire implements FOOL by translating FOOL formulas into FOL formulas. We described how this translation is done for each of the new features. Furthermore, we described a modification of the superposition calculus by FOOL paramodulation that makes Vampire reasoning in FOOL more efficient. We also gave a simple extension to FOOL that allows one to express the next state relation of a program as a boolean formula which is linear in the size of the program.

Neither FOOL nor polymorphic arrays can be expressed in TFF0. In order to support them Vampire uses a modification of the TFF0 syntax with the following features:

1. the boolean sort `$o` can be used as the sort of arguments and quantifiers;

2. boolean variables can be used as formulas, and formulas can be used as boolean arguments;

3. `if-then-else` expressions are represented using a single keyword `$ite` rather than two different keywords `$ite_t` and `$ite_f`;

4. `let-in` expressions are represented using a single keyword `$let` rather than four different keywords `$let_tt`, `$let_tf`, `$let_ft` and `$let_ff`;

5. `$array`, `$select` and `$store` are used to represent arrays of arbitrary types.

Our experimental results have shown that our implementation, and especially FOOL paramodulation, are efficient and can be used to solve hard problems.

Many program analysis problems, problems used in the SMT community, and problems generated by interactive provers, which previously required (sometimes complex) ad hoc translations to first-order logic, can now be understood by Vampire without any translation. Furthermore, Vampire can be used to translate them to the standard TPTP without `if-then-else` and `let-in` expressions, that is, the format understood by essentially all modern first-order theorem provers and used at recent CASC competitions. One should simply use `-mode preprocess` and Vampire will output the translated problem to `stdout` in the TPTP syntax.

The translation to FOL described here is only the first step to the efficient handling of FOOL. It can be considerably improved. For example, the translation of `let-in` expressions always introduces a fresh function symbol together with a definition for it, whereas in some cases inlining the function would produce smaller clauses. Development of a better translation of FOOL is an important future work.

FOOL can be regarded as the smallest superset of the SMT-LIB 2 Core language and TFF0. A native implementation of an SMT-LIB parser in Vampire is an interesting future work. Note that such an implementation can also be used to translate SMT-LIB to FOOL or to FOL.

Another interesting future work is extending FOOL to handle polymorphism and implementing it in Vampire. This would allow us to parse and prove problems expressed in the TFF1 [11] syntax. Note that the current usage of `$array` conforms with the TFF1 syntax for type constructors.

Acknowledgements

We acknowledge funding from the Austrian FWF National Research Network RiSE S11409-N23, the Swedish VR grant D049770 — GenPro, the Wallenberg Academy Fellowship 2014, and the EPSRC grant “Reasoning in Verification and Security”.

Chapter 4

A Clausal Normal Form Translation for FOOL

Evgenii Kotelnikov, Laura Kovács, Martin Suda and Andrei Voronkov

Abstract. Superposition-based theorem provers for first-order logic usually operate on sets of first-order clauses. It is well-known that the translation of a first-order formula to clausal normal form (CNF) can crucially affect the performance of a prover. This paper presents a superposition-friendly translation of FOOL formulas to first-order clauses. Compared to our previous approach, the new translation can produce a smaller set of clauses with fewer introduced symbols.

4.1 Introduction

Automated theorem provers for first-order logic usually operate on sets of first-order clauses. In order to check a formula in full first-order logic, theorem provers first translate it to clausal normal form (CNF). It is well-known that the quality of this translation affects the performance of the theorem prover. While there is no absolute criterion of what the best CNF for a formula is, theorem provers usually try to make the CNF smaller according to some measure. This measure can include the number of clauses, the number of literals, the lengths of the clauses and the size of the resulting signature, i.e. the number of function and predicate symbols. Implementors of CNF translations commonly employ formula simplification [34], (generalised) formula naming [34, 1], and other clausification techniques, aimed to make the CNF smaller.

Our recent work [28] presented a modification of many-sorted first-order with first-class boolean sort. We called this logic FOOL, standing for first-order logic (FOL) with boolean sort. FOOL extends FOL by (i) treating boolean terms as formulas; (ii) `if-then-else` expressions; and (iii) `let-in` expressions. There is a model-preserving translation of FOOL formulas to FOL that works by replacing parts of a FOOL formula with applications of fresh function and predicate symbols and extending the set of assumptions with definitions of these symbols. We implemented [27] this translation in the Vampire theorem prover [30]. To check a FOOL problem Vampire first translates it to first-order logic, then converts the resulting first-order formulas to a set of clauses.

While the translation from [28] provides an easy way to support FOOL in existing first-order provers, it is not necessarily efficient. A more efficient translation can convert a FOOL formula directly to a set of first-order clauses, skipping the intermediate step of converting it to full first-order logic. This way, the translation can integrate known clausification techniques and improve the quality of the resulting clausal normal form.

In this work we present a clausification algorithm that translates a FOOL formula to an equisatisfiable set of first-order clauses. This algorithm aims to minimise the number of clauses and the size of the resuting signature, especially on formulas with `if-then-else`, `let-in` expressions and complex boolean structure. This ultimately leads to an increase of performance of a theorem prover that implements it compared e.g. to the use of the translation to full first-order logic from [28].

Our algorithm is an extension of the NEW-CNF [35] clausification al-

gorithm[1] for first-order logic that enables it to translate FOOL formulas. Section 4.2 revisits the essentials of NEW-CNF which are required for our extension presented in Section 4.3. Our algorithm combines translation of FOOL formulas to first-order logic and clausification. In Section 4.4 we discuss how integrating clausification techniques help to produce smaller clausal normal forms. Section 4.5 describes the experiments on theorem proving with FOOL formulas using different translations. Finally, Section 4.6 outlines future work.

The main contributions of this paper are the following:

1. a clausification algorithm that translates a FOOL formula to an equisatisfiable set of first-order clauses;

2. an implementation of this algorithm in the Vampire theorem prover;

3. experimental results that demonstrate an increase of performance of Vampire on FOOL problems compared to its version with the translation of FOOL formulas to first-order logic presented in [28].

4.2 Clausal Normal Form for First-Order Logic

Traditional approaches to clausification [34] produce a clausal normal form of a given first-order formula in several stages, where each stage represents a single pass through the formula tree. These stages usually include (in this order): formula simplification, translation into negation normal form, formula naming, elimination of equivalences, skolemisation, and distribution of disjunctions over conjunctions. NEW-CNF takes a different approach that employs a single top-down traversal of the formula in which these stages are combined. This approach enables optimisations that are not available if the stages of clausification are independent. For example, compared the traditional staged approach NEW-CNF can introduce fewer Skolem functions on formulas with a complex nesting of equivalences and quantifiers.

The main advantage of NEW-CNF for this work, however, is that its top-down traversal provides a suitable context not only for clausification of first-order formulas, but also of the extension of first-order logic with FOOL features.

[1] The name NEW-CNF is tentative and is likely to be changed before the publication of [35].

NEW-CNF works with the input first-order formula represented as a set of *intermediate clauses*. An intermediate clause C_θ is a pair of a multiset C of signed first-order formulas and a substitution θ that maps variables to terms. We denote by ψ^σ a first-order formula ψ, signed with $\sigma \in \{+, -\}$. Signs are used for polarity dependent elimination of equalities [34]. Substitution θ is used for skolemisation. During the translation, θ is extended by skolemised variables and their corresponding Skolem terms.

NEW-CNF starts with the input first-order formula φ and a set D of intermediate clauses that contains a single intermediate clause $\{\varphi^+\}_\epsilon$, where ϵ is an empty substitution. Then it makes a series of replacements of intermediate clauses in D until all intermediate clauses in D contain only signed atomic formulas. A replacement of an intermediate clause might introduce Skolem functions and names of subformulas. Each replacement preserves the following invariant: the input formula is equivalent with respect to the original signature to the conjunction of universally quantified formulas of the form $\bigwedge_{\psi^\sigma \in C} \psi'$ for every C_θ in D, where every ψ' is $\psi\theta$ if $\sigma = +$ and $\neg\psi\theta$ if $\sigma = -$. When every ψ in each intermediate clause is atomic, D contains the representation of a clausal normal form of the input formula.

For every subformula of φ, NEW-CNF maintains its list of occurrences in the intermediate clauses of D. These occurrences are used for naming of formulas and are updated whenever intermediate clauses are added or removed from D.

The replacements of intermediate clauses are guided by the structure of φ. NEW-CNF traverses φ top-down, visiting every non-atomic subformula of φ exactly once in an order that respects the subformula relation. It means that for each distinct subformulas ψ_1 and ψ_2 of φ such that ψ_1 is a subformula of ψ_2, ψ_2 is visited before ψ_1.

For every subformula ψ, NEW-CNF computes its number of occurrences in intermediate clauses in D. If this number exceeds a pre-specified naming threshold, the formula ψ is named as follows. Let $y_1, \ldots, y_n$ be free variables of ψ and $\tau_1, \ldots, \tau_n$ be their sorts. NEW-CNF introduces a new predicate symbol P of the sort $\sigma_1 \times \ldots \times \sigma_n$. Then, each occurrence ψ^σ in intermediate clauses in D is replaced by $P(y_1, \ldots, y_n)^\sigma$. Finally, two intermediate clauses $\{P(y_1, \ldots, y_n)^-, \psi^+\}_\epsilon$ and $\{P(y_1, \ldots, y_n)^+, \psi^-\}_\epsilon$ are added to D. If the number of occurrences of ψ does not exceed the naming threshold, each of the intermediate clauses that have an occurrence of ψ^+ or ψ^- is replaced with one or more new intermediate clauses according to the rules, described below.

Let ψ be a subformula of φ and C_θ be an intermediate clause such that C has an occurrence of ψ^σ. The intermediate clauses that are added to D depend on the top-level connective of ψ. For $\sigma = +$ we have the following rules. The rules for $\sigma = -$ are dual.

- Suppose that ψ is of the form $\neg\gamma$. Add an intermediate clause to D obtained from C by replacing the occurrence of ψ^+ with γ^-.
- Suppose that ψ is of the form $\gamma_1 \vee \gamma_2$. Add an intermediate clause to D obtained from C by replacing the occurrence of ψ^+ with ${\gamma_1}^+, {\gamma_2}^+$.
- Suppose that ψ is of the form $\gamma_1 \wedge \gamma_2$. Add two intermediate clauses to D obtained from C by replacing the occurrence of ψ^+ with ${\gamma_1}^+$ and ${\gamma_2}^+$, respectively.
- Suppose that ψ in of the form $\gamma_1 \Leftrightarrow \gamma_2$. Add two intermediate clauses to D obtained from C by replacing the occurrence of ψ^+ with ${\gamma_1}^+, {\gamma_2}^-$ and ${\gamma_1}^-, {\gamma_2}^+$, respectively.
- Suppose that ψ in of the form $\gamma_1 \not\Leftrightarrow \gamma_2$. Add two intermediate clauses to D obtained from C by replacing the occurrence of ψ^+ with ${\gamma_1}^+, {\gamma_2}^+$ and ${\gamma_1}^-, {\gamma_2}^-$, respectively.
- Suppose that ψ is of the form $(\forall x : \tau)\gamma$. Add an intermediate clause obtained from C by replacing the occurrence of ψ^+ with γ^+.
- Suppose that ψ is of the form $(\exists x : \tau)\gamma$. Let $y_1, \ldots, y_n$ be all free variables of ψ and $\tau_1, \ldots, \tau_n$ be their sorts. Introduce a fresh Skolem function symbol sk of the sort $\tau_1, \ldots, \tau_n \to \tau$. Add an intermediate clause ${C'}_{\theta'}$, where C' is obtained from C by replacing the occurrence of ψ^+ with γ^+, and θ' extends θ with $x \mapsto sk(y_1, \ldots, y_n)$.

When all subformulas of φ are traversed and the respective rules of replacing intermediate clauses are applied, the set D only contains intermediate clauses with signed atomic formulas. D is then converted to a set of first-order clauses by applying the substitution of each intermediate clause to its respective formulas.

Whenever an intermediate clause C_θ is constructed, NEW-CNF eliminates immediate tautologies and redundant formulas. It means that

1. if C contains both ψ^+ and ψ^-, C_θ is not added to D;
2. if C contains multiple occurrences of a formula with the same sign, only one occurrence is kept in C;
3. if C contains $\top^+$ or $\bot^-$, C_θ is not added to D;

4. if C contains $\bot^+$ or $\top^-$, it is not kept in C.

These rules are not required for replacing intermediate clauses, however they simplify formulas and make the resulting set of clauses smaller.

4.3 Clausal Normal Form for FOOL

This section presents a clausification algorithm for FOOL. This algorithm takes a FOOL formula as input and produces a set of first-order clauses. The conjunction of these clauses is equisatisfiable to the input formula.

FOOL extends many-sorted first-order logic with an interpreted boolean sort and the following syntactical constructs:

1. boolean variables used as formulas;
2. formulas used as arguments to function and predicate symbols;
3. `if-then-else` expressions that can occur as terms and formulas;
4. `let-in` expressions that can occur as terms and formulas and can define an arbitrary number of function and predicate symbols.

There are several ways to support the interpreted boolean sort in a first-order logic. The approach taken in [28] proposes to axiomatise it by adding two constants *true* and *false* of this sorts and two axioms: $\mathit{true} \neq \mathit{false}$ and $(\forall x : \mathit{bool})(x \doteq \mathit{true} \vee x \doteq \mathit{false})$. Furthermore, [28] proposed a modification of superposition calculus that included a replacement of the second axiom with the specialised FOOL paramodulation rule. This modification prevents possible performance problems of a superposition theorem prover caused by self-paramodulation of $x \doteq \mathit{true} \vee x \doteq \mathit{false}$. The translation to first-order clauses presented in this section does not require boolean axioms or modifications of superposition calculus to correctly support the boolean sort. This property is explained at the end of this section.

Our algorithm is an extension of NEW-CNF that adds support for FOOL formulas. In order to enable NEW-CNF to translate FOOL and not just first-order formula we make the following changes to it.

- We allow intermediate clauses to contain signed FOOL formulas, and not just first-order formulas.
- We extend the NEW-CNF tautology elimination with the support for boolean variables. Whenever a boolean variable occurs in an intermediate clause twice with the opposite signs, that intermediate

clause is not added to D. Whenever a boolean variable occurs in an intermediate clause multiple times with the same sign, only one occurrence is kept in the intermediate clause.

- We add extra rules that guide how intermediate clauses are replaced in the set D, detailed below. These rules correspond to syntactical constructs available in FOOL but not in ordinary first-order logic.

- We change the rule that translates existentially quantified formulas to skolemise boolean variables using Skolem predicates and not Skolem functions. For that, we also allow substitutions to map boolean variables to Skolem literals.

- We add an extra step of translation. After the input formula has been traversed, we apply substitutions of boolean variables to every formula in each respective intermediate clause. The resulting set of intermediate clauses might have Skolem literals occurring as terms. We run the clausification algorithm again on this set of intermediate clauses. The second run does not introduce new substitutions and results with a set of intermediate clauses that only contains atomic formulas and substitutions of non-boolean variables.

We extend the rules of replacing intermediate clauses with the cases detailed below. We will not distinguish formulas used as arguments as a separate syntactical construct, but rather treat each such formula φ as an `if-then-else` expression of the form `if` φ `then` *true* `else` *false*. We will assume that every `let-in` expression defines exactly one function or predicate symbol. Every `let-in` expression that defines more that one symbol can be transformed to multiple nested `let-in` expressions, each defining a single symbol, possibly by renaming some of the symbols. Moreover, we will assume that `let-in` expressions only occur as formulas. Every formula that contains a `let-in` expression that occurs as non-boolean term can be transformed to a `let-in` expression that defines the same symbol and occurs as formula.

Let ψ be a subformula of the input formula φ and C_θ be an intermediate clause such that C has an occurrence of ψ^σ.

- Suppose that ψ is a boolean variable x. If θ does not map x, add the intermediate clause ${C'}_{\theta'}$ to D, where C' is obtained from C by removing the occurrence of ψ^σ and θ' extends θ with $x \mapsto \mathit{false}$ if $\sigma = +$, and $x \mapsto \mathit{true}$ if $\sigma = -$. If θ does map x, add C_θ to D.

- Suppose that ψ is $\gamma_1 \doteq \gamma_2$, where γ_1 and γ_2 are formulas. Add

two intermediate clauses to D obtained from C by replacing the occurrence of ψ with ${\gamma_1}^{-\sigma}$, ${\gamma_2}^{+}$ and ${\gamma_1}^{\sigma}$, ${\gamma_2}^{-}$, respectively.

- Suppose that ψ is `if` χ `then` γ_1 `else` γ_2. Add two intermediate clauses to D obtained from C by replacing the occurrence of ψ^σ with χ^-, ${\gamma_1}^{\sigma}$ and χ^+, ${\gamma_2}^{\sigma}$, respectively.

- Suppose that ψ is an atomic formula that contains one or more `if-then-else` expressions occurring as terms. Each of the `if-then-else` expressions is translated in one of two ways, either by expanding or by naming. We will describe both ways for a single `if-then-else` expressions and then generalise for an arbitrary number of `if-then-else` expressions. Suppose that ψ is an atomic formula $L[\texttt{if}\ \gamma\ \texttt{then}\ s\ \texttt{else}\ t]$.

  Expanding. Add two intermediate clauses to D obtained from C by replacing the occurrence of φ^σ with γ^-, $L[s]^\sigma$ and γ^+, $L[t]^\sigma$, respectively.

  Naming. Let $x_1, \ldots, x_n$ be all free variables of φ, and $\tau_1, \ldots, \tau_n$ be their sorts. Let τ be be the sort of both s and t. Then,

  1. introduce a fresh predicate symbol P of the sort $\tau \times \tau_1 \times \ldots \times \tau_n$;
  2. introduce a fresh variable y of the sort τ;
  3. add an intermediate clause to D that is obtained from C by replacing the occurrence of ψ^σ with $L[y]^\sigma$, $P(y, x_1, \ldots, x_n)^-$;
  4. add intermediate clauses $\{\gamma^-, P(s, x_1, \ldots, x_n)^+\}_\epsilon$ and $\{\gamma^+, P(t, x_1, \ldots, x_n)^+\}_\epsilon$ to D.

  In order to eliminate all `if-then-else` expressions we apply either expanding or naming to each of the `if-then-else` expressions. We assume that a pre-specified expansion threshold limits the maximal number of expanded `if-then-else` expressions inside one atomic formula. We start by expanding all `if-then-else` expression and once the expansion threshold is reached, name the remaining `if-then-else` expressions.

- Suppose that ψ is $\texttt{let}\ f(x_1 : \sigma_1, \ldots, x_n : \sigma_n) = t\ \texttt{in}\ \gamma$. It is translated in one of two ways, either by inlining or by naming. The choice of inlining or naming of `let-in` expressions in the problem is determined by a pre-specified boolean option.

Inlining. Add an intermediate clause to D that is obtained from C by replacing the occurrence of ψ^σ with ${\gamma'}^\sigma$. γ' is obtained from γ by replacing each application $f(t_1, \ldots, t_n)$ of a free occurrence of f in γ with t', that is obtained from t by replacing each free occurrence of $x_1, \ldots, x_n$ in t with $t_1, \ldots, t_n$, respectively. We point out that inlining predicate symbols of zero arity does not hinder identification of tautologies thanks to intermediate tautology removal inside intermediate clauses.

Naming. Add an intermediate clause to D that is obtained from C by replacing the occurrence of ψ^σ with γ^σ. Let τ be the sort of t. If τ is *bool*, add intermediate clauses $\{f(x_1, \ldots, x_n)^-, t^+\}_\epsilon$ and $\{f(x_1, \ldots, x_n)^+, t^-\}_\epsilon$ to D. Otherwise, add an intermediate clause $\{f(x_1, \ldots, x_n) \doteq t\}_\epsilon$ to D.

The extra step of translation that eliminates Skolem literals occurring as terms amounts to application of the expansion threshold-based procedure for `if-then-else` expressions.

The extended NEW-CNF algorithm produces a set of first-order clauses. This set does not require boolean axioms to be equisatisfiable to the original FOOL formula. The resulting set of clauses has the following two properties.

1. It can only contain boolean variables and constants *true* and *false* as boolean terms. Every boolean term that occurs in φ is translated as formula and no boolean terms other than variables, *true* and *false* are introduced.

2. It does not contain equalities between boolean terms. Every boolean equality occurring in the input is translated as equivalence between its arguments, and no new boolean equalities are introduced.

These two properties ensure that boolean variables will only be unified with *true* and *false* during superposition. Constants *true* and *false* cannot be unified with each other, therefore no logical inference can violate the properties of the boolean sort.

4.4 Discussion of the Translation

Our extended NEW-CNF algorithm translates FOOL formulas to sets of first-order clauses. It can be used in first-order theorem provers to support reasoning in FOOL. This algorithm combines translation of FOOL

to first-order logic and clausification. This allowed us to enhance the translation by integrating clausification techniques into it. In particular, our extension integrates skolemisation, formula naming and tautology elimination.

In what follows we look at the translation of different features of FOOL done by the extended NEW-CNF and point out how the integrated clausification techniques help to obtain smaller clausal normal forms. We compare the extended NEW-CNF with our translation of FOOL formulas to full first-order logic presented in [28].

4.4.1 Boolean Variables

Our translation of FOOL formulas to full first-order logic replaces each boolean variable x occurring as formula with $x \doteq \mathit{true}$ and skolemises boolean variables using boolean Skolem functions. The extended NEW-CNF skolemises boolean variables using Skolem predicates and substitutes boolean variables that do not need skolemisation with constants *true* and *false*. The approach taken in NEW-CNF is superior in two regards.

1. The translation of FOOL to full first-order logic converts each skolemised boolean variable x occurring as formula to an equality between Skolem terms and *true*. This translation requires a modification of superposition calculus presented in [28] in order to avoid possible performance problems during superposition. NEW-CNF converts x to a Skolem literal, that can be efficiently handled by standard superposition.

2. Substitution of a universally quantified boolean variable with *true* and *false* can decrease the size of the translation. If the variable occurs as formula, after applying the substitution, either the occurrence or the intermediate clause altogether will be discarded by tautology elimination.

We note that the extended NEW-CNF translates formulas in quantified boolean form (QBF) to a clausal normal form of effectively propositional logic (EPR). Every literal in this translation is a Skolem predicate applied to boolean variables and constants *true* and *false*.

4.4.2 `if-then-else`

The extended NEW-CNF translates `if-then-else` expressions occurring as formulas and as terms differently.

An `if-then-else` expression that occurs as formula is translated by introducing two intermediate clauses with a copy of the condition in each one. Translation of a nesting of such `if-then-else` expressions easily leads to an exponential increase in the number of intermediate clauses. This is however averted by the formula naming mechanism of NEW-CNF.

An `if-then-else` expression that occurs as term is translated either by expansion or naming. Expansion doubles the number of intermediate clauses with an occurrence of the condition of `if-then-else`, and does not introduce fresh symbols. Naming adds exactly two new intermediate clauses but introduces a fresh symbol. The expansion threshold provides a trade-off between the increase of the number of intermediate clauses and the number of introduced symbols. For a large number of `if-then-else` expressions it avoids the exponential increase in the number of intermediate clauses. For a small number of `if-then-else` expressions inside an atomic formula it avoids growing the signature.

Formula naming averts the exponential increase in the number of intermediate clauses caused by expansion of nested `if-then-else` expressions that occurs as terms. Consider for example the TPTP problem `SYO500^1.003` that contains a conjecture of the form

$$f_0(f_1(f_1(f_1(f_2(x))))) \doteq f_0(f_0(f_0(f_1(f_2(f_2(f_2(x))))))),$$

where f_0, f_1 and f_2 are unary predicates that take a boolean argument and x is a boolean constant. The extended NEW-CNF translates as an `if-then-else` expression each application of f_i that occurs as argument. Expansion of every `if-then-else` expression doubles the number of intermediate clauses. However, the growth stops once the naming threshold is reached.

Our translation of FOOL formulas to full first-order logic replaces each non-boolean `if-then-else` expression with an application of a fresh function symbol and adds the definition of the symbol to the set of assumptions. The definition is expressed as equality. The extended NEW-CNF avoid introducing new equalities and uses predicate guards for naming. This avoid possible performance problems caused by self-paramodulation similar to the ones described in [28].

4.4.3 `let-in`

Our translation of FOOL formulas to full first-order logic always name `let-in` expressions. The extended NEW-CNF provides the option to either name or inline `let-in` expressions. Naming introduces a fresh function or predicate symbol and does not multiply the resulting clauses. Inlining,

on the other hand, does not introduce any symbols, but can drastically increase the number of clauses. Either of the translation might make a theorem prover inefficient. We point out that the number of clauses and the size of the resulting signature are not the only factors in that. For example, consider inlining of a `let-in` expression that defines a term. It does not introduce a fresh function symbol and does not increase the number of clauses. However, the inlined definition might increase the size of the term with respect to the simplification ordering. This affects the order in which literals will be selected during superposition, and ultimately the performance of the prover.

Designing a syntactical criteria for choosing between naming and inlining is an interesting task for future work.

4.5 Experimental Results

Vampire is the first theorem prover to implement NEW-CNF. We extended Vampire's implementation of NEW-CNF to enable support for FOOL formulas. This extension comprised about 500 lines of C++ code.

In this section we present experimental results obtained by running Vampire on FOOL problems. In particular, we compare performance of Vampire with the extended NEW-CNF algorithm and with the translation of FOOL formulas to FOL presented in [28]. In the sequel, by Vampire we will mean its version with the extended NEW-CNF. We will write Vampire$\star$ for its version with the translation of FOOL formulas to FOL and enabled FOOL paramodulation.

For our experiments we used two sets of problems. The first set is taken from our previous work [27] on the implementation of FOOL in Vampire. The seconds set consists of problems from the SMT-LIB library [7], a corpus of benchmarks for satisfiability modulo theory (SMT) solvers.

In our previous work [27] we experimented with our initial implementation of FOOL in Vampire. For that experiment we generated two set of FOOL problems.

1. Problems from the higher-order part of the TPTP library [42] that can be directly expressed in FOOL. We translated these problems from the TPTP language of higher-order logic to the modification of TPTP that supports FOOL.

2. Problems about properties of (co)algebraic datatypes generated by the Isabelle theorem prover [33] to be checked by SMT solvers. We

translated these problems from the SMT-LIB 2 language to TPTP using the SMTtoTPTP tool [8].

For this work we used the second set of problems and run Vampire in the matching experimental setup. We did not use the first set in this work — problems in this set are easy and all of them were already solved by Vampire before.

Our results are summarised in Tables 4.1–4.2 and discussed below.

4.5.1 Experiments with Algebraic Datatypes Problems

The set of problems about (co)algebraic datatypes generated by Isabelle and translated by us to the TPTP syntax contains 152 problems. All of them use FOOL features: boolean variables occurring as formulas, formulas occurring as arguments to function and predicate symbols, and `if-then-else` expressions. None of the 152 problems use `let-in` expressions.

We run Vampire twice on each problem: once using the option `--mode casc`, and once using `--mode casc_sat`. Both times the `if-then-else` expansion threshold was set to 3, the default value. For each problem, we recorded the fastest successful run of Vampire. We then compared Vampire with the results of Vampire$\star$, CVC4 [6] and Z3 [20], taken from [27].

Table 4.1 summarises the results of our experiments on these 152 problems. Vampire solved the largest number of problems, and all problems solved by Vampire$\star$ were also solved by Vampire. Figure 4.1 shows the Venn diagram of the sets of problems solved by Vampire, CVC4 and Z3, where the numbers denote the numbers of solved problems. Compared to Vampire$\star$, Vampire solved one more problem that was previously only solved by Z3 and 18 more problems, not solved by either Z3 or CVC4. This is significant because the problems were tailored for SMT reasoning. Based on our experimental results we observe that our implementation of the extended NEW-CNF improved the performance of Vampire on this set of problems.

4.5.2 Experiments with SMT-LIB Problems

FOOL can be regarded as a superset of SMT-LIB core logic and problems of SMT-LIB core logic can be directly expressed in FOOL. The language of FOOL extends the SMT-LIB core language with local function definitions, using `let-in` expressions defining functions of arbitrary, and not just zero,

Table 4.1. Runtimes in seconds of provers on the set of 152 algebraic datatypes problems.

Prover	Solved	Total time on solved problems
Vampire	78	19.416
Vampire ⋆	59	26.580
Z3	57	4.291
CVC4	53	25.480

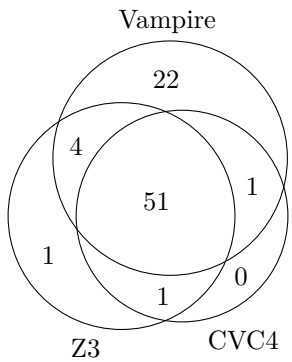

Figure 4.1. Venn diagram of the subsets of the algebraic datatypes problems, solved by Vampire, CVC4 and Z3.

arity. A theorem prover that supports FOOL can be straightforwardly extended to read problems written in the SMT-LIB syntax.

For this experiment we used problems in quantified predicate logic with uninterpreted functions stored in the UF subspace of the SMT-LIB library. These problems are written in the SMT-LIB 2 syntax. In order to read them we implemented a parser for a sufficient subset of the SMT-LIB 2 language in Vampire. The implementation comprised about 2,500 lines of C++ code.

In this experiment we evaluated performance of Vampire, Vampire ⋆, and CVC4 on unsatisfiable problems of the UF subspace. Each problem in the SMT-LIB library is marked with one of the statuses `sat`, `unsat` and `unknown`. A problem is marked as `sat` or `unsat` when at least two SMT solved proved it to be satisfiable or unsatisfiable, respectively. Otherwise, a problem is marked as `unknown`. In order to filter out satisfiable problems we run Vampire, Vampire ⋆, and CVC4 on the problems marked as `unsat` and `unknown` and then recorded the results on the problems that were proven unsatisfiable by at least one prover. That gave us 2596 problems.

Table 4.2. Runtimes in seconds of provers on the set of 2596 unsatisfiable SMT-LIB problems.

Prover	Solved	Total time on solved problems
Vampire	2329	11,057.374
CVC4	2084	26,309.466
Vampire ⋆	2060	14,189.568

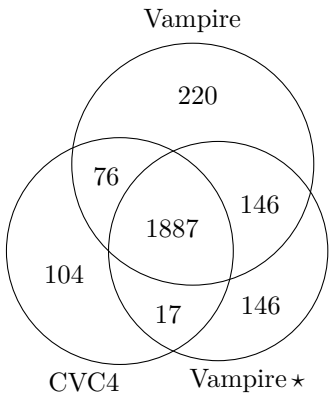

Figure 4.2. Venn diagram of the subsets of the unsatisfiable SMT-LIB problems, solved by Vampire, Vampire ⋆ and CVC4.

The problems in this set use `if-then-else` expressions, `let-in` expressions that define constants, and formulas occurring as arguments to equality. None of the problems use quantifiers over the boolean sort.

We run Vampire twice on each problem: once with naming of `let-in` expressions, and once with inlining. Both times the `if-then-else` expansion threshold was set to 3, the default value. In both runs we also used the option `--mode casc`. For each problem, we recorded the fastest successful run of Vampire. We run Vampire ⋆ once on each problem with the option `--mode casc`.

Table 4.2 summarises the results of our experiments on the SMT-LIB problems. These results are obtained on the StarExec compute cluster [41] using the time limit of 5 minutes per problem. Vampire solved the largest number of problems and was the fastest among the provers. None of the provers solved a superset of problems solved by another prover. Figure 4.2 shows the Venn diagram of the sets of problems solved by Vampire, Vampire ⋆, and CVC4, where the numbers denote the

numbers of solved problems. Vampire solved 296 problem not solved by Vampire$\star$, and Vampire$\star$ solved 163 problems not solved by Vampire. Moreover, we recorded how different translations of `let-in` affected the performance of Vampire. Vampire with inlining of `let-in` expressions solved 314 problems not solved by Vampire without inlining of `let-in` expressions. Vampire without inlining of `let-in` expressions solved 95 problems not solved by Vampire without inlining of `let-in` expressions.

Based on the results of this experiment we make the following observations. Vampire solved new problems by inlining `let-in` expressions and expanding `if-then-else` expressions. Vampire could not solve some of the problems that were solved by Vampire$\star$, likely because of expanding of `if-then-else` rather than naming. Both inlining and naming of `let-in` expressions can make a prover inefficient.

4.6 Conclusion and Future Work

We presented a clausification algorithm for FOOL. It takes a FOOL formula as input and produces an equisatisfiable set of first-order clauses. Our algorithm is based on the NEW-CNF clausification algorithm for first-order logic and extends it to support FOOL formulas.

Our algorithm aims to minimise the number of clauses and the size of the resulting signature it produces. It combines translation of FOOL to first-order logic and clausification. This combination allowed us to integrate into the translation clausification techniques such as skolemisation, formula naming and tautology elimination.

We implemented the extended NEW-CNF algorithm in the Vampire theorem prover. Our experimental results showed an increase of performance of Vampire compared to its version with the translation of FOOL formulas to full first-order logic. We observed that new problems can be solved by expansion of `if-then-else` and instantiation of boolean variables with boolean constants. We observed that both inlining and naming of `let-in` expressions can make a theorem prover succeed or fail.

For future work we are interested in developing syntactical criteria that determine whether a given `let-in` or `if-then-else` expression should be named or inlined, or expanded or inlined, respectively.

Bibliography

[1] Noran Azmy and Christoph Weidenbach. Computing tiny clause normal forms. In *Automated Deduction – CADE-24*, pages 109–125. Springer, 2013.
— One citation on page 67

[2] Leo Bachmair and Harald Ganzinger. On Restrictions of Ordered Paramodulation with Simplification. In *Proceesing of the 10th International Conference on Automated Deduction, Kaiserslautern, FRG, July 24-27, 1990*, pages 427–441, 1990.
— One citation on page 3

[3] Leo Bachmair and Harald Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *Journal of Logic and Computation*, 4(3):217–247, 1994.
— One citation on page 3

[4] Leo Bachmair and Harald Ganzinger. Resolution Theorem Proving. In *Handbook of Automated Reasoning*, pages 19–99. Elsevier and MIT Press, 2001.
— One citation on page 26

[5] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. The Coq proof assistant reference manual: Version 6.1. 1997.
— One citation on page 5

[6] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Proceedings of CAV*, pages 171–177, 2011.
— 4 citations on pages 1, 14, 60, and 78

[7] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. Technical report, Department of Computer Sci-

ence, The University of Iowa, 2010. Available at www.SMT-LIB.org.
— 7 citations on pages 7, 17, 28, 35, 59, 62, and 77

[8] Peter Baumgartner. SMTtoTPTP — A Converter for Theorem Proving Formats. In *Proceedings of CADE*, volume 9195 of *LNCS*, pages 285–294, 2015.
— 3 citations on pages 59, 62, and 78

[9] Christoph Benzmüller, Larry Paulson, Frank Theiss, and Arnaud Fietzke. LEO-II — A Cooperative Automatic Theorem Prover for Higher-Order Logic. In *Proceedings of IJCAR*, volume 5195 of *LNAI*, pages 162–170, 2008.
— 2 citations on pages 1 and 58

[10] Armin Biere. Lingeling, plingeling, picosat and precosat at SAT race 2010. *FMV Report Series Technical Report*, 10(1), 2010.
— One citation on page 1

[11] Jasmin C. Blanchette and Andrei Paskevich. TFF1: The TPTP Typed First-Order Form with Rank-1 Polymorphism. In *Proceedings of CADE*, volume 7898 of *LNCS*, pages 414–420, 2013.
— 5 citations on pages 4, 28, 34, 62, and 63

[12] Sascha Böhme and Tobias Nipkow. Sledgehammer: Judgement Day. In *Proceedings of IJCAR*, pages 107–121, 2010.
— 2 citations on pages 5 and 16

[13] Maria Paola Bonacina. On theorem proving for program checking: historical perspective and recent developments. In *Proceedings of the 12th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 26-28, 2010, Hagenberg, Austria*, pages 1–12, 2010.
— One citation on page 4

[14] Chad E. Brown. Satallax: An Automated Higher-Order Prover (System Description). In *Proceedings of IJCAR*, volume 7364 of *LNAI*, pages 111–117, 2012.
— 2 citations on pages 1 and 58

[15] Alan Bundy. A survey of automated deduction. In *Artificial intelligence today*, pages 153–174. Springer, 1999.
— One citation on page 1

[16] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, 1936.
— One citation on page 2

[17] Martin Davis. The early history of automated deduction. *Handbook of Automated Reasoning*, 1:3–15, 2001.
— One citation on page 1

[18] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.
— One citation on page 3

[19] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM (JACM)*, 7(3):201–215, 1960.
— One citation on page 3

[20] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Proceedings of TACAS*, volume 4963 of *LNCS*, pages 337–340, 2008.
— 4 citations on pages 1, 14, 60, and 78

[21] Ioan Dragan and Laura Kovács. Lingva: Generating and Proving Program Properties Using Symbol Elimination. In *Proceedings of PSI*, pages 67–75, 2014.
— One citation on page 15

[22] Ashutosh Gupta, Laura Kovács, Bernhard Kragl, and Andrei Voronkov. Extensionality Crisis and Proving Identity. In *Proceedings of ATVA*, volume 8837 of *LNCS*, pages 185–200, 2014.
— 4 citations on pages 4, 14, 33, and 49

[23] John Harrison. A short survey of automated reasoning. In *Algebraic Biology*, pages 334–349. Springer, 2007.
— One citation on page 1

[24] Thomas Hillenbrand and Christoph Weidenbach. Superposition for Bounded Domains. In *Automated Reasoning and Mathematics — Essays in Memory of William W. McCune*, pages 68–100, 2013.
— 2 citations on pages 29 and 61

[25] Kryštof Hoder, Laura Kovács, and Andrei Voronkov. Playing in the grey area of proofs. In *Proceedings of POPL*, pages 259–272, 2012.
— 2 citations on pages 4 and 13

[26] Konstantin Korovin. iProver — An Instantiation-Based Theorem Prover for First-Order Logic (System Description). In *Proceedings of IJCAR*, pages 292–298, 2008.
— 2 citations on pages 1 and 13

[27] Evgenii Kotelnikov, Laura Kovács, Giles Reger, and Andrei Voronkov. The Vampire and the FOOL. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs, 2016*, pages 37–48, 2016.
— 3 citations on pages 67, 77, and 78

[28] Evgenii Kotelnikov, Laura Kovács, and Andrei Voronkov. A First Class Boolean Sort in First-order Theorem Proving and TPTP. In *Intelligent Computer Mathematics*, pages 71–86. Springer, 2015.
— 12 citations on pages 33, 35, 36, 37, 54, 61, 67, 68, 71, 75, 76, and 77

[29] Laura Kovács and Andrei Voronkov. Finding Loop Invariants for Programs over Arrays Using a Theorem Prover. In *Proceedings of FASE*, volume 5503 of *LNCS*, pages 470–485, 2009.
— 4 citations on pages 4, 13, 47, and 55

[30] Laura Kovács and Andrei Voronkov. First-Order Theorem Proving and Vampire. In *Proceedings of CAV*, volume 8044 of *LNCS*, pages 1–35, 2013.
— 6 citations on pages 1, 13, 26, 33, 47, and 67

[31] K. L. McMillan. Quantified Invariant Generation Using an Interpolating Saturation Prover. In *Proceedings of TACAS*, pages 413–427, 2008.
— 2 citations on pages 4 and 13

[32] Robert Nieuwenhuis and Albert Rubio. Paramodulation-Based Theorem Proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 7, pages 371–443. Elsevier Science, 2001.
— 2 citations on pages 3 and 26

[33] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. 2002.
— 5 citations on pages 5, 16, 35, 58, and 77

[34] Andreas Nonnengart and Christoph Weidenbach. Computing small clause normal forms. *Handbook of Automated Reasoning*, 1:335–367, 2001.
— 3 citations on pages 67, 68, and 69

[35] Giles Reger, Martin Suda, and Voronkov Andrei. NEW-CNF: a new clausification algorithm for first-order logic. In preparation.
— 2 citations on pages 67 and 68

[36] Andrew Reynolds and Jasmin C. Blanchette. A Decision Procedure for (Co)datatypes in SMT Solvers. In *Proceedings of CADE*, volume 9195 of *LNCS*, pages 197–213, 2015.
— One citation on page 59

[37] George Robinson and Larry Wos. Paramodulation and theorem-proving in first-order theories with equality. *Machine intelligence*, 4:135–150, 1969.
— 2 citations on pages 4 and 27

[38] John Alan Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.
— One citation on page 3

[39] Stephan Schulz. System Description: E 1.8. In *Proceedings of LPAR*, volume 8312 of *LNCS*, pages 735–743, 2013.
— 4 citations on pages 1, 13, 26, and 58

[40] Niklas Sorensson and Niklas Een. Minisat v1.13 – a SAT solver with conflict-clause minimization. *SAT*, 2005:53, 2005.
— One citation on page 1

[41] Aaron Stump, Geoff Sutcliffe, and Cesare Tinelli. StarExec: A cross-community infrastructure for logic solving. In *Automated Reasoning – 7th International Joint Conference, IJCAR 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 19-22, 2014. Proceedings*, pages 367–373, 2014.
— One citation on page 80

[42] Geoff Sutcliffe. The TPTP Problem Library and Associated Infrastructure. *Journal of Automated Reasoning*, 43(4):337–362, 2009.
— 7 citations on pages 4, 14, 28, 33, 34, 58, and 77

[43] Geoff Sutcliffe. Proceedings of the CADE-25 ATP System Competition CASC-25. Technical report, University of Miami, US, 2015. `http://www.cs.miami.edu/~tptp/CASC/25/Proceedings.pdf`.
— One citation on page 58

[44] Geoff Sutcliffe and Christoph Benzmüller. Automated reasoning in higher-order logic using the TPTP THF infrastructure. *Journal of*

Formalized Reasoning, 3(1):1–27, 2010.
— One citation on page 41

[45] Geoff Sutcliffe, Stephan Schulz, Koen Claessen, and Peter Baumgartner. The TPTP Typed First-Order Form with Arithmetic. In *Proceedings of LPAR*, volume 7180 of *LNCS*, pages 406–419, 2012.
— 3 citations on pages 4, 7, and 33

[46] Geoff Sutcliffe and Christian Suttner. The State of CASC. *AI Communications*, 19(1):35–48, 2006.
— One citation on page 4

[47] Andrzej Trybulec. Mizar. In *The Seventeen Provers of the World, Foreword by Dana S. Scott*, pages 20–23, 2006.
— 2 citations on pages 5 and 16

[48] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *J. of Math*, 58(345-363):5, 1936.
— One citation on page 2

[49] Josef Urban, Kryštof Hoder, and Andrei Voronkov. Evaluation of Automated Theorem Proving on the Mizar Mathematical Library. In *ICMS*, pages 155–166, 2010.
— 2 citations on pages 5 and 16

[50] Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. Spass version 3.5. In *CADE*, pages 140–145, 2009.
— One citation on page 26

[51] Larry Wos, George A. Robinson, Daniel F. Carson, and Leon Shalla. The Concept of Demodulation in Theorem Proving. *J. ACM*, 14(4):698–709, 1967.
— One citation on page 4