CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Thesis for the Degree of Doctor of Philosophy

Automated Theorem Proving with Extensions of First-Order Logic

Evgenii Kotelnikov

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

Gothenburg, Sweden
2018

Automated Theorem Proving with Extensions of First-Order Logic

Evgenii Kotelnikov

ISBN 978-91-7597-770-6

Doktorsavhandlingar vid Chalmers tekniska högskola
Ny serie nr. 4451
ISSN 0346-718X

Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Sweden
Telephone +46 (0)31-772 1000

Gothenburg, Sweden, 2018

Abstract

Automated theorem provers are computer programs that check if a logical conjecture follows from a set of logical statements. The conjecture and the statements are expressed in the language of a formal logic, such as first-order logic. Expressivity of first-order logic makes it convenient for encoding problems from diverse application domains. As a result, theorem provers for first-order logic have been used for automation in proof assistants, verification of programs, static analysis of networks, and other purposes. However, their efficient usage remains challenging. One of the challenges is the complexity of translating domain problems to the logic of theorem provers. Not only can such translation be cumbersome due to semantic differences between the domain and the logic, but it might inadvertently result in problems that provers cannot easily handle.

The work presented in the thesis addresses this challenge by developing an extension of first-order logic named FOOL. FOOL contains syntactical features of programming languages and more expressive logics, is friendly for translation of problems from various domains, and can be efficiently supported by existing theorem provers. We describe the syntax and semantics of FOOL and present a simple translation from FOOL to plain first-order logic. We describe a more efficient clausal normal form transformation algorithm for FOOL and based on it implement a support for FOOL in the Vampire theorem prover. We illustrate the efficient use of FOOL for program verification by describing a concise encoding of next state relations of imperative programs in FOOL. We show a usage of features of FOOL in problems of static analysis of networks. We demonstrate the efficiency of automated theorem proving in FOOL with an extensive set of experiments. In these experiments we compare the performance of Vampire on a large collection of problems from various sources translated to FOOL and ordinary first-order logic. Finally, we fix the syntax for FOOL in TPTP, the standard language of first-order theorem provers.

Acknowledgements

I am indebted to many people who in different ways supported me during these past five years.

I would like to thank my supervisor Laura Kovács for setting me on the path that eventually led to this thesis and her help along the way. I am grateful to my co-supervisor Andrei Voronkov for his guidance and for showing me how academic work should be approached. I want to thank my local co-supervisor Moa Johansson for always being ready to help and my examiner Koen Claessen for his good advice and timely support. To my collaborators Martin Suda and Giles Reger, thank you for the insightful discussions about automated reasoning that we had. I am grateful to Byron Cook for inviting me to visit Amazon Web Services and for the great time that I had there.

I was lucky to be surrounded by a lot of amazing colleagues and friends at Chalmers. Daniel H., Daniel S., Iulia, Jeff, Mauricio, Raúl, Pablo, Simon and others, thank you for all the amazing time we spent together. Another thank you goes to Carlo, Grischa, Enzo, Marco and Pierguiseppe for sharing my interest in music. I hope we will get to play more gigs in the future.

Finally, a very special thank you goes to Lydia for her continuous support and patience.

This work has been supported by the ERC Starting Grant 2014 SYMCAR 639270, the Wallenberg Academy Fellowship 2014 and the Swedish VR grant D0497701.

Contents

Introduction

This thesis studies automated theorem proving in first-order logic and its applications. The history of automated theorem proving in first-order logic dates back to the early 1950s (see e.g. [22, 26, 37] for a historical overview). Over the years proof search algorithms and implementations of automated theorem provers have matured and are now used for practical applications. Among these applications are static analysis and verification of software and hardware, automation for proof assistants, knowledge representation, natural language processing and others.

The efficient usage of first-order theorem provers might be challenging. One of the challenges is representation of application problems in first-order logic in a way that is efficient for automated reasoning. Systems that rely on first-order provers, such as program verification tools and proof assistants, usually do not deal with first-order logic natively. 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. There could be multiple ways of translating a problem because of the mismatch between the semantics of the domain and that of first-order logic. A theorem prover might succeed on the results of some of these translations and fail on the others. Users of a theorem prover might find designing a translation that is friendly to the prover to be a difficult task. Such translation might require solid knowledge of how theorem provers work and are implemented, something that the users of the prover might not have. Assessing whether a translation of a certain problem to first-order logic is good might be difficult as well. Such assessment can often only be made through tedious experiments with running theorem provers, configured with different settings, on the results of the translation. A perfect translation might not necessarily exist, because different translation might work better in different scenarios. Furthermore, for some types of problems, their translations to first-order logic cannot be efficiently handled by a theorem prover at all unless the prover is extended with specialised inference rules and heuristics.

The complexity of preparing problems for first-order theorem provers can be battled by extending the logic supported by the provers. Such extension should include theories and new syntactical features that are common in problems from application domains but sensitive to translations. The appropriate translation of these features to plain first-order logic therefore becomes the responsibility of the provers themselves. The right choice of new features and their efficient implementation in theorem provers facilitates applications of automated theorem proving. Firstly, users of theorem provers are relieved from the tedious translations and can express their problems closer to their original domains. Secondly, theorem provers are able to implement translations of these features that suit them best. Thirdly, theorem provers can try multiple different translations in the same proof attempt. Finally, theorem provers can enhance proof search for problems with specific features by implementing dedicated inference rules and preprocessing steps for these features.

This thesis addresses the following research question: *which new extensions of first-order theorem provers are useful for applications and how can these extensions be efficiently implemented?* The thesis identifies that first class boolean sort, `if-then-else` and `let-in` expressions are useful for problems from program verification and automation of proof assistants and are generally not supported by first-order theorem provers. The thesis presents a modification of first-order logic named FOOL that contains these features and gives new techniques for reasoning in it and using it. The thesis describes implementation details and challenges in the Vampire theorem prover, however the described extensions and their implementation can be carried out in any other first-order prover.

This chapter describes the background of the thesis and is structured as follows. First, we overview the key concepts of automated theorem proving in first-order logic. Then, we explain how program verification tools and proof assistants benefit from extensions of theorem provers presented in the thesis. Finally, we detail the main contributions of the thesis and overview its structure.

Automated Theorem Proving in First-Order Logic

First-order logic is not decidable, there is no algorithm that could in general determine whether a given first-order formula is valid or not. First-order logic is semi-decidable, an algorithm that enumerates all finite derivations in the logical system until a given first-order formula is found, terminates if the formula is valid, and may run forever otherwise. If a

formula is satisfiable but not valid, there is no algorithm that could in general demonstrate that. A well studied and generally best performing class of algorithms that search for validity of first-order problems are those based on *saturation* and the calculus of *resolution* and *superposition*. These algorithms are implemented in automated theorem provers such as E [75], iProver [46] and Vampire [54].

First-order theorem provers work with first-order formulas represented as sets of *clauses*. A first-order formula is in a clausal normal form (CNF) if it is a universally quantified conjunction of disjunctions of literals. An alternative representation of a CNF is as a set of first-order clauses, where each clause is a finite multiset of literals. A *clausification* algorithm converts an arbitrary first-order formula to a set of first-order clauses, preserving satisfiability. First-order provers that support formulas in full first-order logic implement such algorithms as part of their preprocessing of the input.

First-order theorem provers construct proofs by *refutation*. Given a first-order problem of the form $\mathit{Premises} \Rightarrow \mathit{Conjecture}$, a theorem prover first negates the conjecture, obtaining $\mathit{Premises} \wedge \neg \mathit{Conjecture}$, then converts this formula to a set of clauses S and attempts to show that S is unsatisfiable by deriving contradiction (the empty clause). To that end, the theorem prover *saturates* the set S with respect to some *inference system* $\mathcal{I}$ which is a collection of *inference rules*. An inference rule is a n-ary ($n \geq 0$) relation on clauses written as

$$\frac{A_1 \quad \ldots \quad A_{n-1}}{B} \ ,$$

where $A_1, \ldots, A_{n-1}$ are premises and B is the conclusion. A set of clauses is called saturated with respect to $\mathcal{I}$ if for every inference of $\mathcal{I}$ with premises in this set, the conclusion of the inference also belongs to that set. To saturate the set S, the theorem prover systematically and exhaustively applies inference rules from $\mathcal{I}$ to premises from S and adds the conclusion of each inference to S. If the empty clause is derived during this process, then the initial set S is unsatisfiable and the input problem is valid. In such case the theorem prover returns the proof of the problem as a tree of inferences with clauses from the initial set S as leafs and the empty clause as the root. If after applying all inferences between clauses in the saturated set S the empty clause has not been derived and the inference system $\mathcal{I}$ is complete then the initial set S is satisfiable and the problem is not valid. In such case the theorem prover returns the saturated set S. Saturation might not terminate on a satisfiable set of clauses, in such

case the theorem prover sooner or later runs out of resources and fails. In practice, finite saturation is rare and theorem provers focus on deriving the empty clause by implementing various techniques and heuristics that make exploration of the search space of clauses more efficient.

Modern theorem provers employ inference systems that include refinements of the calculus of resolution, derived from the work of Robinson [74], and superposition, derived from the work of Bachmair and Ganzinger [3] (see also [4, 63]). The inference rules in this calculus are guarded with side conditions which determine whether a rule can be applied. These conditions prevent the search space of clauses from growing too fast and are essential in practice. The key concepts used in these conditions are a *simplification ordering* and a *literal selection function*. They are understood as parameters of the calculus. A simplification ordering on terms $\succ$ captures the notion of simplicity (see e.g. [28]) i.e. $t_1 \succ t_2$ implies that t_2 is in some way simpler than t_1. There are direct extensions of simplification ordering to literals and clauses. A literal selection function determines for a given clause which literals should be used for inferences. Figure 1 shows the most important inference rules of the superposition and resolution calculus (selected literals are underlined). In this figure, mgu denotes a most general unifier of two first-order terms and $L[s]$ ($t[s]$) denotes that a term s occurs in a literal L (term t).

An important concept related to saturation in *redundancy elimination*. A clause C from a set S is called redundant in S if it is a logical consequence of clauses in S strictly smaller than C w.r.t. to a simplification ordering. Redundant clauses can be eliminated from the search space without compromising completeness. A powerful criterion of redundancy of a clause is *subsumption*. A clause A subsumes B if some subclause of B is an instance of A. If a clause A from a set S subsumes B, B is redundant in S. *Saturation up to redundancy* [63] terminates when the inference system cannot derive any new clauses that are not redundant in the search space.

Another powerful technique is *splitting* [41] of long clauses into smaller ones with disjoint sets of used variables so that the search space can be explored in smaller parts. This technique is motivated by the observation that long clauses slow down saturation based proof search. A recent improvement of splitting is the AVATAR architecture [93] that employs a SAT or SMT solver to guide splitting decisions.

The aforementioned notions and methods and many other refinements of proof search, implemented in theorem provers, aim to constrain the growth of the search space and avoid unnecessary inferences. Ultimately,

Resolution

$$\frac{\underline{A} \vee C_1 \quad \underline{\neg A'} \vee C_2}{(C_1 \vee C_2)\theta}\ ,$$

Factoring

$$\frac{\underline{A} \vee A' \vee C}{(A \vee C)\theta}\ ,$$

where, for both inferences, $\theta = \mathrm{mgu}(A, A')$ and A is not an equality

Superposition

$$\frac{\underline{l \doteq r} \vee C_1 \quad \underline{L[s]} \vee C_2}{(L[r] \vee C_1 \vee C_2)\theta}\ ,$$

where $\theta = \mathrm{mgu}(l, s)$, $t\theta \not\succeq s\theta$ and $L[r]$ is not an equality

or

$$\frac{\underline{l \doteq r} \vee C_1 \quad \underline{t[s] \otimes t'} \vee C_2}{(t[r] \otimes t' \vee C_1 \vee C_2)\theta}\ ,$$

where $\theta = \mathrm{mgu}(l, s)$, $t\theta \not\succeq s\theta$, $t'\theta \not\succeq t[s]\theta$ and $\otimes$ is either $\doteq$ or $\not\doteq$

Equality resolution

$$\frac{\underline{s \doteq t} \vee C}{C\theta}\ ,$$

where $\theta = \mathrm{mgu}(s, t)$

Equality factoring

$$\frac{\underline{s \doteq t} \vee s' \doteq t' \vee C}{(t \not\doteq t' \vee s' \doteq t' \vee C)\theta}\ ,$$

where $\theta = \mathrm{mgu}(s, t)$, $t\theta \not\succeq s\theta$ and $t'\theta \not\succeq s'\theta$

Figure 1. The inference rules of the superposition and resolution calculus.

the behaviour of a theorem prover can be tuned in many different ways. Whether or not a theorem prover solves the input problem depends to a large degree on the choice of parameters of the proof search algorithm. Different combinations of these parameters can solve different problems. For that reason theorem provers such as E, iProver and Vampire implement *portfolios* of proof search strategies. Based on certain characteristics of the input, theorem provers select the appropriate strategies and schedules for them, and then run these strategies one by one in a time-slicing fashion. Some of these strategies are designed to be refutationally incomplete — they cannot derive the empty clause from an arbitrary unsatisfiable set of clauses, but for some unsatisfiable sets of clauses they derive the empty clause very quickly. The usage of multiple proof search strategies in the same proof attempt allows theorem provers to succeed on a larger number of problems. Some provers also extend their portfolios with proof search techniques other than saturation. For example, Vampire includes in its portfolios an implementation of the Inst-Gen calculus [47] and a finite model builder [69].

Another contributing factor to the success of a theorem prover is how well the input problem is prepared to be processed by saturation. First-order theorem provers are known to be fragile with respect to the input. Multiple, often subtle, characteristics of a first-order problem might affect the performance of saturation based proof search. These characteristics include, for example, the number of clauses in the problem, the size of clauses and the size of the signature. Theorem provers implement elaborate preprocessing techniques, in particular improvements of clausification algorithms (see e.g. [65, 2, 67]), that aim to produce good sets of clauses.

Some first-order formulas can be problematic for efficient proof search. A common technique employed by theorem provers is to replace such formulas with specialised inference rules. A well known example of this technique is handling of equality. Equality can be finitely axiomatised in first-order logic as a congruence relation. However, resolution and factoring with equality axioms are known to generate a lot of (mostly unnecessary) new clauses and thus is very inefficient. Rather than axiomatising equality, first-order provers consider it part of the logic and implement specialised inference rules for equality reasoning. These inference rules include refinements of the paramodulation rule [97, 73]. They are part of the standard arsenal of inference rules used by theorem provers. Another example is the extensionality resolution rule, implemented in Vampire [36]. This rule replaces difficult extensionality axioms that are routinely used in encodings of data collections and sets.

The performance of first-order theorem provers is evaluated empirically on large corpora of problems. Comparison of provers is mostly based on success rates and run times. The main corpus is the Thousands of Problems for Theorem Provers (TPTP) library [79]. The problems in this corpus are written in a variety of languages, such as FOF for untyped first-order formulas, TFF0 [86] for typed monomorphic first-order formulas and TFF1 [18] for typed rank-1 polymorphic first-order formulas. The TPTP library is used as a basis for the annual CASC system competition [88].

Many practical problems tackled by theorem provers are expressed in the combination of first-order logic and theories. For example, problems coming from program verification routinely use integer arithmetic, arrays and datatypes. Most interesting theories do not have a complete encoding in first-order logic and require dedicated support in theorem provers. Vampire handles the theory of integer arithmetic by (i) automatically adding incomplete relevant theory axioms to the search space; (ii) applying dedicated inference rules for ground evaluation of theory terms; and (iii) using AVATAR modulo theories [66]. Vampire supports

the polymorphic theory or arrays by automatically instantiating theory axioms for each sort of arrays [48]. Finally, Vampire supports datatypes and codatatypes [17]. Their underlying theory of term algebras cannot be finitely axiomatised in first-order logic, however complete reasoning with this theory was implemented using dedicated inference rules.

Extensions of First-Order Logic for Applications

Deductive Program Verification. The task of a program verification tool is to check whether a given program satisfies its specification. A program specification can be expressed with logical formulas that annotate program statements and capture their properties. Typical examples of such properties are pre-conditions, post-conditions and loop invariants. These program properties are checked using various tools (see e.g. [20] for a detailed overview). Deductive program verification sees compliance with specification as a logical problem that can be checked by automated 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 premises and program properties as the conjecture. Validity of the theorem is interpreted as that the program statements have their annotated properties. Conversely, failure to show validity might indicate a bug in the program. Program verification frameworks such as Boogie [6], Why3 [31] and Frama-C [45] rely on automated theorem provers for checking program 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 [61, 53, 40] 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 [53]; such properties could not be generated fully automatically by any previously known method.

Automation of Proof Assistants. Proof assistants are software tools that assist users in constructing proofs of mathematical problems. Proof assistants use formalisations of mathematics based on higher-order logic (Isabelle/HOL [64]), type theory (Coq [7]), set theory (Mizar [90]) 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 [19, 92]. Automation using external theorem provers, including first-order ones, is implemented e.g. in the Sledgehammer extension [19] of Isabelle. Sledgehammer heuristically picks lemmas and definitions that might be necessary for the proof, translates them to the logic of automated theorem provers and hands over the resulting formulas to the provers. If one of the provers returns a proof, Sledgehammer uses this proof to reconstruct a proof in the calculus of Isabelle. The translation of Isabelle's lemmas and definitions might be incomplete because the logic of Isabelle is more expressive than that of automated provers.

The translation of the following features of programming languages and more expressive logics to plain first-order logic might be cumbersome and inefficient. This thesis presents features of FOOL that can be used for a more straightforward translation. Further, the thesis present methods of efficient support of these features and an implementation of these methods in Vampire.

1. Boolean values in programming languages are used both as expressions in conditional and 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 one cannot mix boolean terms and formulas in the same way in plain first-order logic. FOOL contains the boolean sort as its first class sort. Formulas in FOOL are indistinguishable from boolean terms which coincides with the treatment of booleans in programming languages.

2. Properties expressed in higher-order logic routinely use quantification over the interpreted boolean sort; this is not allowed in plain first-order logic. FOOL allows quantification over the first class boolean sort. Besides proof assistants, the first class boolean sort is useful to higher-order automated theorem provers such as Satallax [21] and Leo-II [14] that employ first-order provers for their proof search.

3. Imperative programs are structured as sequences of variable updates. Standard techniques for translating such sequences to logic involve computing a static single assignment (SSA) form of the program. Computation of an SSA form introduces intermediate variables and their presence in the resulting formula can deteriorate the performance of a theorem prover. FOOL contains `let-in` expressions. One can concisely express sequences of variable assign-

ments in FOOL as nested `let-in` and leave the decision of naming intermediate states of the program or not to the theorem prover.

4. Both programming language and logics of proof assistants routinely use conditional expressions and local definitions of functions. The standard approaches for translating them are inlining and naming. Either one of these approaches can result in difficult first-order formulas. FOOL contains `if-then-else` expressions and allows `let-in` expressions to define function and predicate symbols with arbitrary arity. The choice between inlining and naming is left to the theorem prover itself which is better equipped to make it.

Contributions of the Thesis

In summary, the work presented in this thesis

1. introduces the extension FOOL of first-order logic that contains useful syntactical constructs that are usually not supported by first-order provers, mentioned before;

2. explores how reasoning in FOOL can be efficiently implemented in existing automated theorem provers for first-order logic;

3. gives practical evidence of usefulness of FOOL for application through examples and developed translation techniques;

4. gives practical evidence of efficiency of reasoning with FOOL through experimental results on large diverse collections of problems.

FOOL. The thesis presents FOOL, standing for first-order logic (FOL) with boolean sort. FOOL extends ordinary many-sorted FOL with (i) first class boolean sort, (ii) boolean variables used as formulas, (iii) formulas used as arguments to function and predicate symbols, (iv) `if-then-else` expressions and (v) `let-in` expressions. `if-then-else` and `let-in` expressions can occur as both terms and formulas. `let-in` expressions can use (multiple simultaneous) definitions of function symbols, predicate symbols, and tuples. The thesis presents the definition of FOOL, its semantics, and a simple model-preserving translation from FOOL formulas to formulas of first-order logic. This translation can be used to support FOOL in existing first-order provers.

Reasoning with FOOL. The thesis presents two approaches to an implementation of FOOL in first-order provers that improve over the simple translation of FOOL to FOL. The first approach is a new technique of dealing with the boolean sort in superposition theorem provers. This technique includes replacement of one of the boolean sort axioms with a specialised inference rule called FOOL paramodulation. The second approach is a new algorithm $\text{VCNF}_{\text{FOOL}}$ that transforms FOOL formulas directly to first-order clauses. The thesis presents an implementation of the simple translation from FOOL to FOL and both improved approaches in Vampire.

Applications of FOOL. The thesis presents an encoding of the next state relations of imperative programs in FOOL. Compared to similar methods, this encoding avoids introducing intermediate variables and results in FOOL formulas that concisely represent the structure of program fragments in logic. The thesis presents a work on verification of virtual private cloud network configurations with Vampire. The encoding of verification problems in this work relies on first class booleans, the theory of arrays and the theory of tuples.

Practical Evaluation. The thesis presents extensive experiments on running Vampire, other first-order theorem provers, higher-order theorem provers and SMT solvers on FOL and FOOL problems. These problems come from various sources: benchmarks from the TPTP and SMT-LIB library, proof obligations generated by the Isabelle proof assistant, and verification conditions generated by multiple different program verification tools. The experimental results obtained with these problems show in particular that

1. Vampire with FOOL paramodulation performs better than Vampire with the simple translation from FOOL to FOL;
2. Vampire with $\text{VCNF}_{\text{FOOL}}$ performs better that Vampire with FOOL paramodulation;
3. Vampire performs better on verification conditions translated to FOOL than the same verification conditions translated to FOL using methods implemented in state-of-the-art verification tools.

Impact on TPTP. The language of FOOL is a superset of TFF0 — the monomorphic first-order part of the TPTP language. The thesis describes a modification of the TPTP language needed to represent FOOL formulas.

This modification has been included in the TPTP standard as the TPTP Extended Typed First-Order Form (TFX).

Impact on Vampire. The language of FOOL is a superset of the core theory of the SMT-LIB language [10], the standard language of SMT solvers. 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. Vampire gained support for SMT-LIB based on its implementation of FOOL, and since 2016 has been participating in the SMT-COMP competition [11] where it contends against SMT solvers.

The support of both FOOL and theories such as arithmetic, arrays and datatypes, makes Vampire a convenient and powerful tool for reasoning about properties of programs.

Structure of the Thesis

The work described in this thesis has been carried out in six papers, each contained in a separate chapter. Four papers (Chapters 1, 2, 3 and 4) were published in peer-reviewed conferences, one (Chapter 6) was published in a peer-reviewed workshop, and one (Chapter 5) is a technical report not yet submitted for publication. 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 chapters of this thesis are arranged in the order in which their correspondent papers were written. Chapter 1 presents the syntax and semantics of FOOL. Chapter 2 presents the implementation of FOOL in Vampire. Chapter 3 presents an efficient clausification algorithm for FOOL. Chapter 4 describes an encoding of the next state relations of imperative programs in FOOL. Chapter 5 describes an approach to network verification based on automated reasoning in first-order logic, which uses features of FOOL. Finally, Chapter 6 describes TFX, the extension of the TPTP language that contains the syntax for FOOL.

Each of the papers contained in this thesis has been written and presented separately. As a result, the introductory remarks and preliminaries of some of the chapters overlap. Another consequence is that some ideas presented in earlier chapters are revisited and developed in later chapters. One example of such idea is the encoding of the next state relations of

imperative programs in FOOL. A sketch of this encoding first appears in Chapter 2 and preliminary experimental results are discussed in Chapter 3. The precise formal description of the encoding and extensive evaluation is however given later in Chapter 4. Another example is the set of syntactical constructs available in FOOL. The original description of FOOL in Chapter 1 does not include `let-in` expressions with simultaneous definitions, definitions of tuples and tuple expressions. These constructs are included in later chapters.

The contributions of the thesis are the cumulative contributions of all six papers. The rest of this section details the main contributions of each individual paper.

Chapter 1. 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.

Bibliographic information. The paper has been published in the proceedings of the 8th Conference on Intelligent Computer Mathematics (CICM) in 2015 [50].

Chapter 2. 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.

Bibliographic information. The paper has been published in the proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs (CPP) in 2016 [48].

Chapter 3. 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 resulting 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. We extended Vampire with new preprocessing options that can be used to strengthen its portfolios.

Statement of contribution. The paper is co-authored with Laura Kovács, Martin Suda and Andrei Voronkov. Evgenii Kotelnikov contributed with the extension of VCNF that supports FOOL, the implementation of this extension in Vampire and the experiments.

Bibliographic information. The paper has been published in the proceedings of the 2nd Global Conference on Artificial Intelligence (GCAI) in 2016 [49].

Chapter 4. A FOOLish Encoding of the Next State Relations of Imperative Programs

The paper describes an encoding of the next state relations of imperative programs with variable updates and `if-then-else` statements in FOOL. Based on this encoding the paper presents a translation of imperative programs annotated with their pre- and post-conditions to partial correctness properties of these programs. We demonstrate by experiments that this translation results in formulas that are easier for Vampire than the formulas produced by program verification tool such Boogie and BLT.

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

Bibliographic information. The paper has been published in the proceedings of the 9th International Joint Conference on Automated Reasoning (IJCAR) in 2018 [51].

Chapter 5. Checking Network Reachability Properties by Automated Reasoning in First-Order Logic

The paper describes an approach for static verification of virtual private cloud networks using automated theorem proving for first-order logic. We model networks with Horn clauses and check first-order properties of these models using the Vampire theorem prover. We used Vampire both as a saturation-based theorem prover and a finite model builder for different kinds of checked properties.

Statement of contribution. The chapter is co-authored with Pavle Subotić and based on a joint work with Byron Cook, Temesghen Kahsai and Sean McLaughlin. Evgenii Kotelnikov contributed with the encoding of network reachability properties in first-order logic and the implementation of a checker for these problems based on Vampire.

Chapter 6. TFX: The TPTP Extended Typed First-Order Form

The paper presents the new language TFX that extends and simplifies the language of typed first-order formulas TFF. TFX includes the first class

boolean sort, `if-then-else` expressions, `let-in` expressions and tuples. The inclusion of these syntactic constructs was motivated by the work on FOOL and FOOL formulas can be directly expressed in TFX. TFX has been included in the latest release of the TPTP library.

Statement of contribution. The paper is co-authored with Geoff Sutcliffe. Evgenii Kotelnikov contributed with the discussion of the TFX syntax, the description of FOOL and examples of FOOL problems.

Bibliographic information. The paper has been published in the proceedings of the 6th Workshop on Practical Aspects of Automated Reasoning (PAAR) in 2018 [85].

Chapter 1

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.

1.1 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 [61, 53, 40] 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 [53]; 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 [46], E [75], and Vampire [54], 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. [36] 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 [79] 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 [27] and CVC4 [8], 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 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 [29], 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 [64], and the Mizar project [90], one can see that first-order theorem provers are the main workhorses behind computer proofs in both cases — see e.g. [19, 92]. 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:

$$\begin{aligned}&(\forall p : \mathit{bool})(\forall l : \mathit{list}_A)(\forall x : A)(\forall y : A)\\ &\quad \mathtt{contains}(l, \mathrm{ite}(p, x, y)) \doteq\\ &\qquad (p \Rightarrow \mathtt{contains}(l, x)) \wedge (\neg p \Rightarrow \mathtt{contains}(l, y))\end{aligned} \tag{1.1}$$

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

$$\begin{aligned}&(\forall l_1 : \mathit{list}_A)(\forall l_2 : \mathit{list}_A)(\forall p : \mathit{bool})\\ &\neg(\mathtt{subset\_sorted}(l_1, l_2) \doteq p \wedge\\ &\quad (\forall l_2' : \mathit{list}_A)\neg(l_1 \doteq \mathtt{nil} \wedge l_2 \doteq l_2' \wedge p) \wedge\\ &\quad (\forall x_1 : A)(\forall l_1' : \mathit{list}_A)\neg(l_1 \doteq \mathtt{cons}(x_1, l_1') \wedge l_2 \doteq \mathtt{nil} \wedge \neg p) \wedge\\ &\quad (\forall x_1 : A)(\forall l_1' : \mathit{list}_A)(\forall x_2 : A)(\forall l_2' : \mathit{list}_A)\\ &\quad \neg(l_1 \doteq \mathtt{cons}(x_1, l_1') \wedge l_2 \doteq \mathtt{cons}(x_2, l_2') \wedge\\ &\qquad p \doteq \mathrm{ite}(x_1 < x_2, \mathit{false},\\ &\qquad\quad \mathrm{ite}(x_1 \doteq x_2, \mathtt{subset\_sorted}(l_1', l_2'),\\ &\qquad\qquad \mathtt{subset\_sorted}(\mathtt{cons}(x_1, l_1'), l_2')))))\end{aligned} \tag{1.2}$$

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 [10], 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 1.2. We further describe how FOOL can be translated to the ordinary many-sorted first-order logic in Section 1.3. Section 1.4 discusses superposition-based theorem proving and proposes a new way of dealing with the boolean sort in it. In Section 1.5 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 1.6 discusses related work and Section 1.7 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.

1.2 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.

1.2.1 Syntax

We assume a countable infinite set of *variables*.

Definition 1.1. *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 1.2. 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 1.1. 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 1.3. 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.

1.2.2 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 1.4. 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 1.5. 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:

$$\mathrm{eval}_I(x) = [\![\, x \,]\!]_I .$$

$$\mathrm{eval}_I(f(t_1, \ldots, t_n)) = [\![\, f \,]\!]_I \, (\mathrm{eval}_I(t_1), \ldots, \mathrm{eval}_I(t_n)).$$

$$\mathrm{eval}_I(s \doteq t) = \begin{cases} 1, \text{ if } \mathrm{eval}_I(s) = \mathrm{eval}_I(t); \\ 0, \text{ otherwise.} \end{cases}$$

$$\mathrm{eval}_I((\forall x : \sigma)\varphi) = \begin{cases} 1, \text{ if } \mathrm{eval}_{I^a_x}(\varphi) = 1 \text{ for all } a \in [\![\, \sigma \,]\!]_I; \\ 0, \text{ otherwise.} \end{cases}$$

$$\mathrm{eval}_I((\exists x : \sigma)\varphi) = \begin{cases} 1, \text{ if } \mathrm{eval}_{I^a_x}(\varphi) = 1 \text{ for some } a \in [\![\, \sigma \,]\!]_I; \\ 0, \text{ otherwise.} \end{cases}$$

$$\mathrm{eval}_I(\texttt{if } \varphi \texttt{ then } s \texttt{ else } t) = \begin{cases} \mathrm{eval}_I(s), \text{ if } \mathrm{eval}_I(\varphi) = 1; \\ \mathrm{eval}_I(t), \text{ otherwise.} \end{cases}$$

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

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 1.2. 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 1.2 implies that any interpretation, which coincides with I on free variables and free function symbols of φ is also a model of φ.

1.3 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 1.1. 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 1.2. 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 1.3. 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 1.4. 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 1.3. 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.

$$\forall(x : bool)(x \doteq true \vee x \doteq false) \wedge true \not\doteq false. \tag{1.3}$$

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

Given a syntactically first-order formula γ, we denote by $fol(\gamma)$ 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 (1.3).

Theorem 1.4. Let η is a type assignment and γ be a syntactically first-order formula such that $\eta \vdash \gamma : bool$.

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 1.3 and 1.4 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.

1.4 Superposition for FOOL

In Section 1.3 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 [54], E [75] and Spass [95], 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

$$x \doteq \mathit{true} \vee x \doteq \mathit{false}, \tag{1.4}$$

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, 63]. 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 [54], 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 [73]:

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

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 (1.4) 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 (1.4) and results of paramodulation from variables applied to them. This will cause a rapid degradation of performance of superposition 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 (1.4). 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 (1.4).

Now the only possible inferences with (1.4) 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 (1.4) 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*.

1.5 TPTP Support for FOOL

The typed monomorphic first-order formulas subset, called TFF0, of the TPTP language [79], 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 (1.1) and (1.2) of Section 1.1.

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 [18]. 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.

1.6 Related Work

Handling boolean terms as formulas is common in the SMT community. The SMT-LIB project [10] defines its core logic as first-order logic extended with the distinguished first-class boolean sort and the `let-in` ex-

pression 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 [38]. The approach of [38] 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 [38] 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.

1.7 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 2

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

2.1 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, [36] 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 implemented in the theorem prover Vampire [54] to natively support mirroring programming language constructs in its input language. They include (i) FOOL [50], 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 2.2 presents how FOOL is implemented in Vampire and focuses on new extensions to the TPTP input language [79] of first-order provers. Section 2.2 extends the TPTP language of monomorphic many-sorted first-order formulas, called TFF0 [86], 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 2.3 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 [18] 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 2.4 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 2.5. 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 2.6 discusses related work and Section 2.7 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 [79], 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 [64]. 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.

2.2 First Class Boolean Sort

Our recent work [50] 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 [10], 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.

2.2.1 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: $(\forall x : bool)(x \doteq true \vee x \doteq false)$ and $true \not\doteq false$. However, as we discuss in [50], using this axiomatisation in a superposition theorem prover may result in performance problems caused by self-paramodulation of $x \doteq true \vee x \doteq false$.

To overcome this problem, in [50] 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 $true \not\doteq false$.

3. Add a special inference rule, called *FOOL paramodulation*, of the form
$$\frac{C[s]}{C[true] \vee s \doteq false},$$
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 [50]. 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.

2.2.2 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)`.

2.2.3 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 as follows:

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

2.2.4 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:

$$(\forall x : \sigma)(\forall y : \tau)(\forall z : \tau) impl(P(x, y) \wedge P(x, z), y \doteq z). \tag{2.1}$$

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*, (2.1) 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. $(\forall x : \sigma)(\forall y : \tau)(\forall z : \tau)(P(x, y) \wedge P(x, z) \Leftrightarrow g_1(x, y, z) \doteq true)$;
2. $(\forall y : \tau)(\forall z : \tau)(y \doteq z \Leftrightarrow g_2(y, z) \doteq true)$.

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 (2.1) 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 $Says(p, s)$ 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 $Knight(p)$ or $Knave(p)$ 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 $(\forall p : person)(\forall s : bool)(Knight(p) \wedge Says(p, s) \Rightarrow s)$ and $(\forall p : person)(\forall s : bool)(Knave(p) \wedge Says(p, s) \Rightarrow \neg s)$, respectively. We will express the fact that every person is either a knight or a knave by the axiom $(\forall p : person)(Knight(p) \oplus Knave(p))$, where $\oplus$ is the "exclusive or" connective. Finally, we will express the statements that Zoye and Mel make in the puzzle by axioms $Says(zoye, Knave(mel))$ and $Says(mel, \neg Knave(zoye) \wedge \neg Knave(mel))$, 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) [84]. 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.

2.2.5 `if-then-else` Expressions

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

$$(\forall x : \mathbb{Z})(\forall y : \mathbb{Z})(max(x, y) \doteq \texttt{if } x \geq y \texttt{ then } x \texttt{ else } y). \tag{2.2}$$

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 (2.2) 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*:

$$(\forall x : \mathbb{Z})(\forall y : \mathbb{Z})(\texttt{if } max(x, y) \doteq x \texttt{ then } x \geq y \texttt{ else } y \geq x). \tag{2.3}$$

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 (2.2) and `$ite_f` in (2.3).

2.2.6 `let-in` Expressions

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

$$\begin{aligned} &\texttt{let } f_1(x_1^1 : \sigma_1^1, \ldots, x_{n_1}^1 : \sigma_{n_1}^1) = s_1; \\ &\qquad \ldots \\ &\qquad f_m(x_1^m : \sigma_1^m, \ldots, x_{n_m}^m : \sigma_{n_m}^m) = s_m \\ &\texttt{ in } t, \end{aligned} \tag{2.4}$$

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 (2.4) is not in general equivalent to m nested `let-ins`

$$\begin{array}{l} \texttt{let } f_1(x_1^1 : \sigma_1^1, \ldots, x_{n_1}^1 : \sigma_{n_1}^1) = s_1 \texttt{ in} \\ \quad \ddots \\ \quad\quad \texttt{let } f_m(x_1^m : \sigma_1^m, \ldots, x_{n_m}^m : \sigma_{n_m}^m) = s_m \texttt{ in} \\ \quad\quad\quad t. \end{array} \tag{2.5}$$

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`.

$$\begin{array}{l} \texttt{let } array(i : \mathbb{Z}) = \texttt{if } i \doteq 3 \texttt{ then } 5 \texttt{ else } array(i) \texttt{ in} \\ \quad array(2) + array(3) \end{array} \tag{2.6}$$

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 $f(b, a)$.

$$\texttt{let } a = b;\ b = a \texttt{ in } f(a, b) \tag{2.7}$$

In order to handle `let-in` expressions Vampire translates them to FOL. This is done in three stages for each expression in (2.4).

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 (2.5). 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 (2.6) 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 (2.7) is translated in the following way. First, the `let-in` expression is translated to the form (2.5). The constant a has a free occurrence in the body of b, therefore it is replaced by a fresh constant a'. The formula (2.7) becomes

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

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.

Formula (2.6) 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 `$let` expression can be used for multiple bindings. For that, the bindings should be separated by a semicolon and passed as the first argument. The formula (2.7) can be written using `$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)))))).
```

2.3 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 [53]. Previous versions of Vampire supported theories of integer arrays and arrays of integer arrays [54]. 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.

2.3.1 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 $\mathit{array}(\tau, \sigma)$, function symbols $\mathit{select} : \mathit{array}(\tau, \sigma) \times \tau \to \sigma$ and $\mathit{store} : \mathit{array}(\tau, \sigma) \times \tau \times \sigma \to \mathit{array}(\tau, \sigma)$, 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

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

2. read-over-write 2

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

3. extensionality

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

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 2.2:

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

Formula (2.6) used an interpreted function to represent the array in this code. We can alternatively use arrays to represent it as follows

$$\begin{array}{l} \texttt{let } \mathit{array} = \mathit{store}(\mathit{array}, 3, 5) \texttt{ in} \\ \quad \mathit{select}(\mathit{array}, 2) + \mathit{select}(\mathit{array}, 3) \end{array} \tag{2.8}$$

2.3.2 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 $\mathit{array}(\tau, \sigma)$, Vampire adds this sort and function symbols *select* and *store* of the types $\mathit{array}(\tau, \sigma) \times \tau \to \sigma$ and $\mathit{array}(\tau, \sigma) \times \tau \times \sigma \to \mathit{array}(\tau, \sigma)$, 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:

$$\begin{array}{c} (\forall a : \mathit{array}(\tau, \sigma))(\forall i : \tau)(\forall v : \sigma) \\ (\mathit{select}(\mathit{store}(a, i, v), i) \doteq v) \end{array} \tag{2.9}$$

$$\begin{array}{c} (\forall a : \mathit{array}(\tau, \sigma))(\forall i : \tau)(\forall j : \tau)(\forall v : \sigma) \\ (i \not\doteq j \Rightarrow \mathit{select}(\mathit{store}(a, i, v), j) \doteq \mathit{select}(a, j)) \end{array} \tag{2.10}$$

$$\begin{array}{c} (\forall a : \mathit{array}(\tau, \sigma))(\forall b : \mathit{array}(\tau, \sigma)) \\ (a \not\doteq b \Rightarrow (\exists i : \tau)(\mathit{select}(a, i) \not\doteq \mathit{select}(b, i))) \end{array} \tag{2.11}$$

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 (2.11) 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 [36] 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 $\mathit{array}(\tau, \sigma)$ 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 corre-

sponding 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 (2.8) can be written in the extended TFF0 syntax as follows:

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

2.3.3 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 $array(\tau, bool) \times \tau \to bool$ and the *store* function has the type $array(\tau, bool) \times \tau \times bool \to array(\tau, bool)$. 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:

$$\begin{aligned}&(\forall a : array(\tau, bool))(\forall i : \tau)(\forall v : bool)\\&\quad(select(store(a, i, v), i) \Leftrightarrow (v \doteq true))\end{aligned}$$

$$\begin{aligned}&(\forall a : array(\tau, bool))(\forall i : \tau)(\forall j : \tau)(\forall v : bool)\\&\quad(i \neq j \Rightarrow select(store(a, i, v), j) \Leftrightarrow select(a, j))\end{aligned}$$

$$\begin{aligned}&(\forall a : array(\tau, bool))(\forall b : array(\tau, bool))\\&\quad(a \neq b \Rightarrow (\exists i : \tau)(select(a, i) \oplus select(b, i)))\end{aligned}$$

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 $array(\mathbb{Z}, bool) \times array(\mathbb{Z}, bool) \to array(\mathbb{Z}, bool)$. 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)))).
```

2.4 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 of 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.

2.4.1 Encoding the Next State Relation

Consider the program given in Figure 2.1, 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 2.1 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 *max*) as a hypothesis in the extended TFF0 syntax of FOOL, by using the

```
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 2.1. 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 := $diff(res, max),
                                 res)),
                res)))).

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

Figure 2.2. Representation of the partial correctness statement of the code on Figure 2.1 in Vampire.

`if-then-else` (`$ite`) and `let-in` (`$let`) constructs, and (ii) the safety property as the conjecture to be proven by Vampire.

Figure 2.2 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 2.1 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;

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

Figure 2.3. Updating multiple variables.

- $[P_1; P_2]$ is $\texttt{let}\ D\ \texttt{in}\ [P_2]$ where $[P_1]$ is $\texttt{let}\ D\ \texttt{in}\ x$.

Given a program P, the next state value for variable x can be given by $[P; \mathrm{x} := \mathrm{x}]$, 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 2.3 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 2.3 can be encoded as follows:

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

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 [50]. 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:

- $[\mathrm{x}_i := e]$ is $\texttt{let}\ (\ldots, x_i, \ldots) = (\ldots, e, \ldots)\ \texttt{in}\ (x_1, \ldots, x_n)$,

- [**if** e **then** P_1 **else** P_2] is `let` $(x_1, \ldots, x_n)$ = `if` e `then` $[P_1]$ `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 2.1. 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.

2.4.2 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 2.4, 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 2.4 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 [53]. In this case we use the FOL property specified in the **invariant** construct of Figure 2.4. 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,

$a \coloneqq 0$; $b \coloneqq 0$; $c \coloneqq 0$;

invariant a = b + c $\wedge$
a $\geq$ 0 $\wedge$ b $\geq$ 0 $\wedge$ c $\geq$ 0 $\wedge$ a $\leq$ 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 $B[b] \coloneqq A[a]$; $b \coloneqq b + 1$;
    else $C[c] \coloneqq A[a]$; $c \coloneqq c + 1$;
  $a \coloneqq a + 1$;
end do

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

Figure 2.4. Array partition.

$a = b + c$), (ii) the number of visited array elements in A, B, C is positive (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 2.4. 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 2.5 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 2.4 is indeed an invariant. Using FOOL+ described in Section 2.4.1, we can verify the inductiveness property of the invariant, as follows: (i) express 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 2.5. Representation of the partial correctness statement of the code on Figure 2.4 in Vampire.

2.5 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 [79] 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 [64], see Subsection 2.5.2 for more details.

Our results are summarised in Tables 2.1–2.3 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/`.

2.5.1 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 latest edition of CASC [82]: Satallax [21], Leo-II [14], and Isabelle [64]. We note that all of them used the first-order theorem prover E [75] for first-order reasoning (Isabelle also used several other provers).

Table 2.1. 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

Table 2.1 summarises our results on these problems. Only Vampire, Vampire ⋆ and Satallax were able to solve all of them, while Vampire was 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.

2.5.2 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 [10]. 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 [70]. To run the benchmark we first translated the SMT-LIB files to the TPTP syntax using the SMTtoTPTP translator [12] 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

Table 2.2. 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

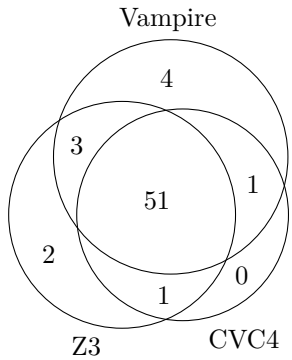


Figure 2.6. Venn diagram of the subsets of the algebraic datatypes problems, solved by Vampire, CVC4 and Z3.

problem, we recorded the fastest successful run of Vampire. We used a similar setting for evaluating Vampire ⋆. In this experiment, we then compared Vampire with the best available SMT solvers, namely with CVC4 [8] and Z3 [27].

Table 2.2 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 2.6 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 2.3 details performance results on these 11 problems.

Based on our experimental results shown in Tables 2.2 and 2.3, 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

Table 2.3. 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

comparable with those of CVC4. Whenever successful, Z3 turned out to be faster than Vampire; we believe this is because of the sophisticated 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.

2.6 Related Work

FOOL was introduced in our previous work [50]. 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 [38]. The approach of [38] 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 [38] 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 [10] 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 [12] 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 [18].

2.7 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 [18] 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 3

A Clausal Normal Form Translation for FOOL

Evgenii Kotelnikov, Laura Kovács, Martin Suda and Andrei Voronkov

Abstract. Automated theorem provers for first-order logic usually operate on sets of first-order clauses. It is well-known that the translation of a formula in full first-order logic to a clausal normal form (CNF) can crucially affect performance of a theorem prover. In our recent work we introduced a modification of first-order logic extended by the first class boolean sort and syntactical constructs that mirror features of programming languages. We called this logic FOOL. Formulas in FOOL can be translated to ordinary first-order formulas and checked by first-order theorem provers. While this translation is straightforward, it does not result in a CNF that can be efficiently handled by state-of-the-art theorem provers which use superposition calculus. In this paper we present a new CNF translation algorithm for FOOL that is friendly and efficient for superposition-based first-order provers. We implemented the algorithm in the Vampire theorem prover and evaluated it on a large number of problems coming from formalisation of mathematics and program analysis. Our experimental results show an increase of performance of the prover with our CNF translation compared to the naive translation.

Published in the *Proceedings of the 2nd Global Conference on Artificial Intelligence*, pages 53–71. EPiC Series in Computing, 2016.

3.1 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 [65], (generalised) formula naming [65, 2], and other clausification techniques, aimed to make the CNF smaller.

Our recent work [50] presented a modification of many-sorted first-order logic with first-class boolean sort. We called this logic FOOL, standing for first-order logic (FOL) with boolean sort. FOOL extends standard FOL by (i) treating boolean terms as formulas, (ii) `if-then-else` expressions, (iii) `let-in` expressions, and (iv) tuple expressions. While `if-then-else` and `let-in` expressions are also available in the SMT-LIB core language [9], the standard input language for SMT solvers, FOOL is a strict superset of SMT-LIB as tuple expressions are not part of SMT-LIB and `let-in` expressions in FOOL can define non-constant functions and predicate symbols.

There is a model-preserving translation of FOOL formulas to FOL (see [50]) 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. To reason about a FOOL formula, one can thus first translate it to a FOL formula and then convert the FOL formula into a set of clauses using the usual first-order clausification techniques. While this translation 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 FOOL to FOL. This way, the translation can integrate known clausification techniques and improve the quality of the resulting clausal normal form.

In this paper we present a new clausification algorithm $\mathrm{VCNF}_{\mathrm{FOOL}}$ that translates a FOOL formula to an equisatisfiable set of first-order clauses. Our algorithm avoids producing large numbers of duplicate clauses and new symbols during clausification and also avoids clauses

that can make theorem provers inefficient. We show that in practice this leads to a significant increase in the performance of a theorem prover).

Our $\text{VCNF}_{\text{FOOL}}$ algorithm is a non-trivial extension of the recent VCNF clausification algorithm for FOL [67]. The extension employs several clausification techniques for handling the features of FOOL, namely boolean terms and `if-then-else`, `let-in` and tuple expressions. These techniques comprise the contributions of this work and are listed below.

Contributions. The main contributions of this paper are the following.

1. We present a new clausification algorithm for translating FOOL formulas to an equisatisfiable set of first-order clauses.

2. We handle boolean variables in FOOL formulas by skolemising them using skolem predicates instead of skolem functions, thus avoiding the introduction of new boolean equalities.

3. We control the clausification of FOOL formulas with `if-then-else` and `let-in` expressions by a threshold level on the number of formula occurrences. Depending on the threshold, our algorithms decides on the fly whether to inline `if-then-else` and `let-in` expressions or introduce a new name and definition for them.

4. We handle tuple expressions in FOOL by introducing so-called projection functions and use these projection functions in the translation of `let-in` expressions with tuple definition.

5. We implemented our work in the Vampire theorem prover [54], offering this way an automated support to reason about FOOL formulas.

6. We evaluate our work on three benchmark suites coming from verification and analysis of software and described in Section 3.4, and show experimentally that our method significantly improves over [48] by the number of solved problems and the runtime.

3.2 Clausal Normal Form for First-Order Logic

Traditional approaches to clausification in FOL [65, 54] produce a clausal normal form in several stages, where each stage represents a single pass through the formula tree. These stages may include formula simplification, translation into (equivalence) negation normal form, formula naming, elimination of equivalences, skolemisation, and distribution of disjunctions

over conjunctions. The VCNF clausification algorithm of [67] takes a different approach and employs a single top-down traversal of the formula in which these stages are combined. This enables optimisations that are not available if the stages of clausification are independent. For example, compared to the traditional staged approach, VCNF can introduce fewer skolem functions on formulas with complex nesting of equivalences and quantifiers. Moreover, it can detect and discard intermediate tautologies, which are much more difficult to recognise by the staged approach.

In this paper we use the VCNF algorithm and extended it to a new clausification algorithm for FOOL [50]. The main advantage of VCNF for our 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. In this section we overview the main features of VCNF. We will follow the notation used in [67] and in what follows will repeat some of the definitions.

3.2.1 Preliminaries

Our setting is that of many-sorted first-order predicate logic with equality.

A signature Σ is a set of *predicate* and *function* symbols together with associated sorts. A *term* of the sort τ is of the form $f(t_1, \ldots, t_n)$, c or x where f is a *function symbol* of the sort $\tau_1 \times \ldots \times \tau_n \to \tau$, $t_1, \ldots, t_n$ are terms of sorts $\tau_1, \ldots, \tau_n$, respectively, c is a constant of sort τ and x is a variable of sort τ. An *atom* is of the form $p(t_1, \ldots, t_n)$, q or $t_1 \doteq t_2$ where p is a *predicate symbol* of the sort $\tau_1 \times \ldots \times \tau_n$, $t_1, \ldots, t_n$ are terms of sorts $\tau_1, \ldots, \tau_n$, respectively, q is a predicate symbol of sort *bool* and $\doteq$ is the *equality symbol*. A literal is an atom or its negation.

A *formula* is of the form $\varphi_1 \wedge \ldots \wedge \varphi_n$, $\varphi_1 \vee \ldots \vee \varphi_n$, $\varphi_1 \Rightarrow \varphi_2$, $\varphi_1 \Leftrightarrow \varphi_2$, $\varphi_1 \not\Leftrightarrow \varphi_2$, $\neg\varphi_1$, $\exists x : \tau.\varphi_1$, $\forall x : \tau.\varphi_1$, $\bot$, $\top$, or l where φ_i are formulas, x is a variable, τ a sort and l is a literal. Note that we treat conjunction and disjunction as n-ary operators; we assume that formulas are kept in *flattened form*, e.g. $(\varphi_1 \wedge \varphi_2) \wedge \varphi_3$ is always represented as $\varphi_1 \wedge \varphi_2 \wedge \varphi_3$. Furthermore, we assume that usage of $\top$ and $\bot$ is simplified immediately.

A *sign* is a either $\mathtt{t}$ or $\mathtt{f}$. A *signed formula* is a pair consisting of a formula φ and a sign $\star \in \{\mathtt{t}, \mathtt{f}\}$, denoted by $\varphi^\star$. The signed formula $\varphi^\mathtt{t}$ (resp. $\varphi^\mathtt{f}$) means that φ is true (resp. false). We will use the mapping $\mathtt{form}$ from signed formulas to formulas defined as follows: $\mathtt{form}(\varphi^\mathtt{t}) = \varphi$ and $\mathtt{form}(\varphi^\mathtt{f}) = \neg\varphi$. We call a *sequent* a finite set of signed formulas. We say that a sequent $S_1, \ldots, S_n$ is true in a FOOL interpretation if so is the universal closure of the formula $\mathtt{form}(S_1) \vee \ldots \vee \mathtt{form}(S_n)$. Note that if

$S_1, \ldots, S_n$ are signed *atomic* FOL formulas, then $\texttt{form}(S_1)\vee\ldots\vee\texttt{form}(S_n)$ is a clause.

3.2.2 VCNF

The VCNF algorithm [67] works with finite sets of sequents. During computation the algorithm may construct substitutions to be applied to existing (signed) formulas. It is convenient for us to collect these substitutions without immediately applying them. For this reason, instead of a sequent $D\theta$, where θ is a substitution, we will use pairs D_θ consisting of a sequent D and a substitution θ. We will (slightly informally) also refer to such pairs as sequents.

The VCNF algorithm starts with the input first-order formula φ and a set C of sequents that contains a single sequent $\{\varphi^{\texttt{t}}\}_\epsilon$, where ϵ is the empty substitution. Then it makes a series of steps replacing sequents in C by other sequents until all sequents in C contain only signed atomic FOL formulas. Some of the steps introduce fresh (previously unused) symbols. Each update of C preserves the following invariants: (1) if an interpretation $\mathcal{I}$ satisfies all sequents after the update, then $\mathcal{I}$ also satisfies all sequents before the update; (2) if an interpretation $\mathcal{I}$ satisfies all sequents before the update, then there exists an interpretation $\mathcal{I}'$ that extends $\mathcal{I}$ on fresh symbols such that $\mathcal{I}'$ satisfies all sequents after the update.

The replacements of sequents are guided by the structure of φ. VCNF traverses φ top-down, processing every non-atomic subformula of φ exactly once in an order that respects the subformula relation. That is, for each two distinct subformulas ψ_1 and ψ_2 of φ such that ψ_1 is a subformula of ψ_2, ψ_2 is processed before ψ_1. For every subformula of φ, VCNF maintains a list of its occurrences as signed formulas in the sequents of C. The occurrences are updated whenever sequents are removed from and added to C. The main role of the list is to allow for a fast enumeration and lookup of all the occurrences when a particular subformula is to be processed. As explained below, the number of occurrences is also used to decided whether a subformula should be named. The replacements are governed by a set of rules that are, essentially, the standard tableau rules for first-order logic. We briefly summarise these rules below, and refer to [67] for details.

We note that except for the rule for negation, which essentially flips the sign of each occurrence of $\psi = \neg\gamma$ and replaces ψ with its immediate sub-formula γ in all the sequents, the remaining rules come in pairs in which they are dual to each other. For instance, dealing with a disjunction

$\gamma_1 \vee \gamma_2$ with a positive $\star = \mathtt{t}$ is analogous to dealing with a conjunction with a negative sign. For simplicity, we only show the versions for $\star = \mathtt{t}$ below.

Let ψ be a subformula of φ and D_θ be a sequent such that D has an occurrence of $\psi^{\mathtt{t}}$. Before proceeding to the next subformula, VCNF visits and replaces all such sequents D. Depending on the top-level connective of ψ the algorithm applies the following rules.

- Suppose that ψ is of the form $\neg\gamma$. Add a sequent to C obtained from D by replacing the occurrence of $\psi^{\mathtt{t}}$ with $\gamma^{\mathtt{f}}$.
- Suppose that ψ is of the form $\gamma_1 \vee \gamma_2$. Add a sequent to C obtained from D by replacing the occurrence of $\psi^{\mathtt{t}}$ with ${\gamma_1}^{\mathtt{t}}, {\gamma_2}^{\mathtt{t}}$.
- Suppose that ψ is of the form $\gamma_1 \wedge \gamma_2$. Add two sequents to C obtained from D by replacing the occurrence of $\psi^{\mathtt{t}}$ with ${\gamma_1}^{\mathtt{t}}$ and ${\gamma_2}^{\mathtt{t}}$, respectively.
- Suppose that ψ in of the form $\gamma_1 \Leftrightarrow \gamma_2$. Add two sequents to C obtained from D by replacing the occurrence of $\psi^{\mathtt{t}}$ with ${\gamma_1}^{\mathtt{t}}, {\gamma_2}^{\mathtt{f}}$ and ${\gamma_1}^{\mathtt{f}}, {\gamma_2}^{\mathtt{t}}$, respectively.
- Suppose that ψ in of the form $\gamma_1 \not\Leftrightarrow \gamma_2$. Add two sequents to C obtained from D by replacing the occurrence of $\psi^{\mathtt{t}}$ with ${\gamma_1}^{\mathtt{t}}, {\gamma_2}^{\mathtt{t}}$ and ${\gamma_1}^{\mathtt{f}}, {\gamma_2}^{\mathtt{f}}$, respectively.
- Suppose that ψ is of the form $(\forall x : \tau)\gamma$. Add a sequent obtained from D by replacing the occurrence of $\psi^{\mathtt{t}}$ with $\gamma^{\mathtt{t}}$.
- Suppose that ψ is of the form $(\exists x : \tau)\gamma$. Let $y_1, \ldots, y_n$ be all free variables of $\psi\theta$ and $\tau_1, \ldots, \tau_n$ be their sorts. Introduce a fresh Skolem function symbol sk of the sort $\tau_1 \times \ldots \times \tau_n \to \tau$. Add a sequent ${D'}_{\theta'}$, where D' is obtained from D by replacing the occurrence of $\psi^{\mathtt{t}}$ with $\gamma^{\mathtt{t}}$, 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 sequents are applied, the set C only contains sequents with signed atomic formulas. C is then converted to a set of first-order clauses by applying the substitution of each sequent to its respective formulas.

Whenever the number of occurrences of a subformula ψ in sequents in C exceeds a pre-specified *naming threshold*, 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. VCNF introduces a new predicate symbol P of the sort $\tau_1 \times \ldots \times \tau_n$. Then, each occurrence $\psi^\star$ in sequents in C is replaced by $P(y_1, \ldots, y_n)^\star$. Finally,

two sequents $\{P(y_1, \ldots, y_n)^{\mathtt{f}}, \psi^{\mathtt{t}}\}_\epsilon$ and $\{P(y_1, \ldots, y_n)^{\mathtt{t}}, \psi^{\mathtt{f}}\}_\epsilon$ are added to C to serve as a definition of ψ. As usual, in case ψ always occurs in C only under a single sign, adding only the one respective defining sequent is sufficient.

Whenever a new sequent D_θ is constructed, VCNF eliminates immediate tautologies and redundant formulas. It means that

1. if D contains both $\psi^{\mathtt{t}}$ and $\psi^{\mathtt{f}}$, D_θ is not added to C;
2. if D contains multiple occurrences of a signed formula, only one occurrence is kept in D;
3. if D contains $\top^{\mathtt{t}}$ or $\bot^{\mathtt{f}}$, D_θ is not added to C;
4. if D contains a signed formula $\bot^{\mathtt{t}}$ or $\top^{\mathtt{f}}$, this signed formula is removed from D.

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

VCNF takes as an input a first-order formula in *equivalence negation normal form* (ENNF). A formula is in ENNF if it does not contain $\Rightarrow$ and negations are only applied to atoms. ENNF is very convenient for standard FOL, as it reduces the number of cases to consider and makes checking polarities trivial. At the same time, it is not easy to define a useful extension of ENNF for FOOL because of `let-in` expressions and formulas inside terms. It is straightforward, however, to extend VCNF in order to support formulas in full first-order logic. For that, we need to add an extra rewriting rule for implications. In what follows we will consider a modification of VCNF with this extension.

3.3 Clausal Normal Form for FOOL

In this section we describe our new clausification algorithm for FOOL. The algorithm takes a FOOL formula as input and produces an equisatisfiable set of first-order clauses. We write $\text{VCNF}_{\text{FOOL}}$ to refer to this algorithm, and FOOL2FOL to refer to the algorithm of [50] for translating FOOL formulas to arbitrary FOL formulas. In what follows, we first briefly overview the FOOL logic and then describe $\text{VCNF}_{\text{FOOL}}$ and compare the CNFs produces by it and FOOL2FOL.

3.3.1 FOOL

FOOL [50] extends the standard many-sorted FOL with an interpreted boolean sort. Boolean variables can be used as formulas in FOOL and

formulas may be used as arguments to function and predicate symbols. In addition to its first-class boolean sort, FOOL extends standard FOL with following constructs:

1. `if-then-else` expressions that can occur as terms and formulas;
2. `let-in` expressions that can occur as terms and formulas and can define an arbitrary number of function and predicate symbols.

Finally, FOOL also includes tuple expressions and `let-in` expressions with tuple definitions. A `let-in` expression with a tuple definition has the form $\texttt{let}\ (c_1, \ldots, c_n) = s\ \texttt{in}\ t$, where $n > 1$, t is a term, $c_1, \ldots, c_n$ are constants, and s is a tuple expression. A tuple expression is inductively defined as follows:

1. $(s_1, \ldots, s_n)$, where $s_1, \ldots, s_n$ are terms;
2. $\texttt{if}\ \varphi\ \texttt{then}\ s_1\ \texttt{else}\ s_2$, where s_1 and s_2 are tuple expressions;
3. a `let-in` expression of the form $\texttt{let}\ D\ \texttt{in}\ t$, where D is tuple, function, or predicate definition, and t is a tuple expression.

Note that tuple expressions are not first class terms. They can only occur on the right-hand side of tuple definitions, but not as arguments to function or predicate symbols. Moreover, we do not assign sorts to tuple expressions and do not allow nested tuple expressions. It is however straightforward to extend FOOL with a theory of first class tuples. For that, one needs to assign tuple sorts of the form $(\tau_1, \ldots, \tau_n)$ to tuple expressions of the form $(s_1, \ldots, s_n)$ if $s_1 : \tau_1, \ldots, s_n : \tau_n$, and allow tuple expression to appear as terms. Such extension is not considered in this paper.

There are several ways to support the interpreted boolean sort in first-order theorem proving. The approach taken in [50] proposes to axiomatise it by adding two constants *true* and *false* of this sorts and two axioms: $\mathit{true} \not\doteq \mathit{false}$ and $(\forall x : \mathit{bool})(x \doteq \mathit{true} \vee x \doteq \mathit{false})$. Furthermore, [50] proposes a modification in superposition calculus of first-order provers: it (i) changes the simplification ordering of first-order prover by making *true* and *false* the smallest terms of boolean sort and (ii) replaces the second axiom with a so-called FOOL paramodulation rule. These modifications block self-paramodulation of $x \doteq \mathit{true} \vee x \doteq \mathit{false}$ and hence prevent performance problems arising from self-paramodulation in superposition theorem proving. In this paper, we however argue that neither boolean axiom nor modifications of superposition calculus are needed to support the interpreted boolean sort. Rather, we propose special processing of

boolean variables and boolean equalities during clausification and avoid the introduction of new boolean equalities.

3.3.2 Introducing $\text{VCNF}_{\text{FOOL}}$

The $\text{VCNF}_{\text{FOOL}}$ clausification algorithm introduced in this paper is a non-trivial extension of the VCNF algorithm. Compared to FOOL2FOL, $\text{VCNF}_{\text{FOOL}}$ clausifies FOOL formulas directly, without first translating them to general FOL formulas and only then to CNF. The $\text{VCNF}_{\text{FOOL}}$ algorithm extends VCNF by adding support for FOOL formulas, as follows.

- We allow sequents to contain signed FOOL formulas, and not just first-order formulas.
- We extend the VCNF tautology elimination with the support for boolean variables. Whenever a boolean variable occurs in a sequent twice with the opposite signs, that sequent is not added to C. Whenever a boolean variable occurs in a sequent multiple times with the same sign, only one occurrence is kept in the sequent.
- We add extra rules that guide how sequents are replaced in the set C 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 sequent. The resulting set of sequents might have skolem literals occurring as terms. We run the clausification algorithm again on this set of sequents. The second run does not introduce new substitutions and results with a set of sequents that only contain atomic formulas and substitutions of non-boolean variables.

In the sequel, we detail the rules of $\text{VCNF}_{\text{FOOL}}$ for replacing sequents. To simplify the exposition and without the loss of generality, we make the following assumptions about the input FOOL formula.

- We do 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 assume that every `let-in` expression defines exactly one function or predicate symbol. Every `let-in` expression that defines more than one symbol can be transformed to multiple nested `let-in` expressions, each defining a single symbol, possibly by renaming some of the symbols.
- We assume that `let-in` expressions only occur as formulas. Every atomic formula that contains a `let-in` expression can be transformed to a `let-in` expression that defines the same symbol and occurs as a formula.
- Finally, we assume that each function or predicate symbol is defined by a `let-in` expression at most once. This can be achieved by a standard renaming policy.

3.3.3 $\mathbf{VCNF_{FOOL}}$ Rules

This section presents the rewriting rules of $\mathrm{VCNF_{FOOL}}$ for syntactic construct available in FOOL, but not in standard first-order logic. For each such construct we present a rewriting rule for it in $\mathrm{VCNF_{FOOL}}$, give an example of a FOOL formula with that construct, and compare its CNFs obtained using $\mathrm{VCNF_{FOOL}}$ and FOOL2FOL.

Let us now fix an input formula φ and let ψ be one of its subformulas. In the sequel we assume that φ and ψ are fixed and give all definitions relative to them. Let D_θ be a sequent such that D has an occurrence of $\psi^\star$.

Boolean Variables

Suppose that ψ is a boolean variable x. If θ does map x, $\mathrm{VCNF_{FOOL}}$ adds D_θ to C. This corresponds to the case in which x was an existentially quantified variable skolemised in some previous step.

If θ does not map x, $\mathrm{VCNF_{FOOL}}$ adds the sequent $D'_{\theta'}$ to C, where D' is obtained from D by removing the occurrence of $\psi^\star$ and θ' extends θ with $x \mapsto \mathit{false}$ if $\star = \mathtt{t}$, and $x \mapsto \mathit{true}$ if $\star = \mathtt{f}$. This corresponds to the case in which x was a universally quantified variable. Treating the boolean universal quantifier as a conjunction, we are implicitly replacing the sequent D with two extensions, one for $x \mapsto \mathit{false}$ and the other for

$x \mapsto \mathit{true}$. However, one of them is always true due to the occurrence of $\psi^\star$ in D and so is not considered anymore. Thus only $D'_{\theta'}$ is further processed by $\mathrm{VCNF}_{\mathrm{FOOL}}$.

Example. Let $\psi_1 = (\forall x : \mathit{bool})(x \vee P(x))$, $\psi_2 = (\exists y : \mathit{bool})(P(y) \wedge y)$, where P is a predicate symbol of the sort $\mathit{bool} \to \mathit{bool}$ and let us consider the formula $\varphi = \psi_1 \vee \psi_2$.

To process φ, $\mathrm{VCNF}_{\mathrm{FOOL}}$ first applies the rule for disjunction inherited from VCNF, obtaining the sequent $\{\psi_1^{\mathrm{t}}, \psi_2^{\mathrm{t}}\}_\epsilon$. The following are the steps corresponding to processing ψ_1 and its subformulas:

$$\begin{array}{ll} \{(\forall x : \mathit{bool})(x \vee P(x))^{\mathrm{t}}, \psi_2^{\mathrm{t}}\}_\epsilon & \Rightarrow \\ \{(x \vee P(x))^{\mathrm{t}}, \psi_2^{\mathrm{t}}\}_\epsilon & \Rightarrow \\ \{x^{\mathrm{t}}, P(x)^{\mathrm{t}}, \psi_2^{\mathrm{t}}\}_\epsilon & \Rightarrow \\ \{x^{\mathrm{t}}, P(x)^{\mathrm{t}}, \psi_2^{\mathrm{t}}\}_{\{x \mapsto \mathit{false}\}}. & \end{array}$$

Notice how the substitution is extended by $x \mapsto \mathit{false}$ because of the positive occurrence of the boolean variable x.

Next, we show how ψ_2 and its subformulas get processed. We introduce sk, a nullary skolem predicate symbol for the existential quantifier:

$$\begin{array}{ll} \{x^{\mathrm{t}}, P(x)^{\mathrm{t}}, (\exists y : \mathit{bool})(P(y) \wedge y)^{\mathrm{t}}\}_{\{x \mapsto \mathit{false}\}} & \Rightarrow \\ \{x^{\mathrm{t}}, P(x)^{\mathrm{t}}, (P(y) \wedge y)^{\mathrm{t}}\}_{\{x \mapsto \mathit{false}, y \mapsto \mathit{sk}\}} & \Rightarrow \\ \{x^{\mathrm{t}}, P(x)^{\mathrm{t}}, P(y)^{\mathrm{t}}\}_{\{x \mapsto \mathit{false}, y \mapsto \mathit{sk}\}}, \{x^{\mathrm{t}}, P(x)^{\mathrm{t}}, y^{\mathrm{t}}\}_{\{x \mapsto \mathit{false}, y \mapsto \mathit{sk}\}}. & \end{array}$$

Recall that dealing with boolean variables in $\mathrm{VCNF}_{\mathrm{FOOL}}$ requires an extra stage in which boolean substitutions are applied:

$$\{\mathit{false}^{\mathrm{t}}, P(\mathit{false})^{\mathrm{t}}, P(\mathit{sk})^{\mathrm{t}}\}_\epsilon, \{\mathit{false}^{\mathrm{t}}, P(\mathit{false})^{\mathrm{t}}, \mathit{sk}^{\mathrm{t}}\}_\epsilon.$$

Next, $\mathrm{VCNF}_{\mathrm{FOOL}}$ eliminates the tautology $\mathit{false}^{\mathrm{t}}$ in both sequents. The literal $P(\mathit{sk})$ contains a formula inside, therefore $\mathrm{VCNF}_{\mathrm{FOOL}}$ translates it as the formula $P(\texttt{if } \mathit{sk} \texttt{ then } \mathit{true} \texttt{ else } \mathit{false})$ according to the rules given in Section 3.3.3:

$$\{P(\mathit{false})^{\mathrm{t}}, P(\texttt{if } \mathit{sk} \texttt{ then } \mathit{true} \texttt{ else } \mathit{false})^{\mathrm{t}}\}_\epsilon, \{P(\mathit{false})^{\mathrm{t}}, \mathit{sk}^{\mathrm{t}}\}_\epsilon.$$

Finally, $\mathrm{VCNF}_{\mathrm{FOOL}}$ converts signed atomic formulas to literals and we obtain the following three clauses:[1]

$$\{P(\mathit{false}), \neg \mathit{sk}, P(\mathit{true})\}, \{P(\mathit{false}), \mathit{sk}\}, \{P(\mathit{false}), \mathit{sk}\}.$$

[1] Notice that the last two clauses are identical and one of them could be dropped. However, $\mathrm{VCNF}_{\mathrm{FOOL}}$ is not designed to do that.

FOOL2FOL converts φ to the following set of clauses:

$$\{x \doteq \mathit{true}, P(x), P(\mathit{sk})\}, \{x \doteq \mathit{true}, P(x), \mathit{sk} \doteq \mathit{true}\},$$

where *sk* is a skolem constant of the sort *bool*. □

The FOOL2FOL algorithm of [50] replaces each boolean variable x occurring as formula with $x \doteq \mathit{true}$ and skolemises boolean variables using boolean skolem functions. Unlike FOOL2FOL, $\mathrm{VCNF}_{\mathrm{FOOL}}$ skolemises boolean variables using skolem predicates and substitutes boolean variables that do not need skolemisation with constants *true* and *false*. The approach taken in $\mathrm{VCNF}_{\mathrm{FOOL}}$ is superior in two regards.

1. FOOL2FOL converts each skolemised boolean variable x occurring as formula to an equality between skolem terms and *true*. $\mathrm{VCNF}_{\mathrm{FOOL}}$ converts x to a skolem literal which can be handled by standard superposition more efficiently.

2. Substitution of a universally quantified boolean variable with *true* and *false* can decrease the size of the translation. If the boolean variable occurs as formula, after applying the substitution, the occurrence is either removed or the whole sequent is discarded by tautology elimination in $\mathrm{VCNF}_{\mathrm{FOOL}}$.

Our treatment of boolean variables never introduces new equalities and uses skolem predicates instead of skolem functions. We process boolean equalities as logical equivalences and use guards to name `if-then-else` expressions occurring as terms. The usage of these techniques give the resulting set of clauses 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 eventually introduced.

These two properties ensure that no extra axioms or inference rules are required to handle the interpreted boolean sort in a theorem prover. In particular, thanks to the second property we do not need any form of equational reasoning for this sort.

Boolean Equalities

Suppose that ψ is $\gamma_1 \doteq \gamma_2$, where γ_1 and γ_2 are formulas. $\mathrm{VCNF}_{\mathrm{FOOL}}$ adds a sequent to C that is obtained from D by replacing the occurrence of $\psi^\star$ with $(\gamma_1 \Leftrightarrow \gamma_2)^\star$.

In effect, $\mathrm{VCNF}_{\mathrm{FOOL}}$ reduces the case of boolean equality to that of formula equivalence, delegating the processing to the respective rule inherited from VCNF.

`if-then-else` Expressions as Terms

Suppose that ψ is an atomic formula that contains one or more `if-then-else` expressions occurring as terms. $\mathrm{VCNF}_{\mathrm{FOOL}}$ translates each of the expressions either by expanding or naming it. We first describe this step of $\mathrm{VCNF}_{\mathrm{FOOL}}$ for a single `if-then-else` expression and then generalise for an arbitrary number of `if-then-else` expressions inside one atomic formula. Suppose that ψ is an atomic formula $L[\texttt{if}\ \gamma\ \texttt{then}\ s\ \texttt{else}\ t]$.

Expanding. $\mathrm{VCNF}_{\mathrm{FOOL}}$ adds two sequents to C obtained from D by replacing the occurrence of $\psi^\star$ with γ^{f}, $L[s]^\star$ and γ^{t}, $L[t]^\star$, respectively.

Naming. Let $x_1, \ldots, x_n$ be all the free variables of γ, and $\tau_1, \ldots, \tau_n$ be their sorts. Let τ be the common sort of both s and t. Then, the $\mathrm{VCNF}_{\mathrm{FOOL}}$ algorithm

1. introduces a fresh predicate symbol P of the sort $\tau \times \tau_1 \times \ldots \times \tau_n$;
2. introduces a fresh variable y of the sort τ;
3. adds a sequent to C that is obtained from D by replacing the occurrence of $\psi^\star$ with $L[y]^\star$, $P(y, x_1, \ldots, x_n)^{\mathrm{f}}$;
4. adds sequents $\{\gamma^{\mathrm{f}}, P(s, x_1, \ldots, x_n)^{\mathrm{t}}\}_\epsilon$ and $\{\gamma^{\mathrm{t}}, P(t, x_1, \ldots, x_n)^{\mathrm{t}}\}_\epsilon$ to C.

Example. Consider a definition of the *max* function using `if-then-else` taken from [48]:

$$(\forall x : \mathbb{Z})(\forall y : \mathbb{Z})(max(x, y) \doteq \texttt{if}\ x \geq y\ \texttt{then}\ x\ \texttt{else}\ y). \tag{3.1}$$

To translate (3.1), $\mathrm{VCNF}_{\mathrm{FOOL}}$ first applies twice the rule for existential quantifier inherited from VCNF, obtaining the sequent

$$\{(max(x, y) \doteq \texttt{if}\ x \geq y\ \texttt{then}\ x\ \texttt{else}\ y)^{\mathrm{t}}\}_\epsilon.$$

Then, either expanding or naming processes the result.

- Expanding results in

$$\{(x \geq y)^{\mathtt{f}}, (max(x,y) \doteq x)^{\mathtt{t}}\}_{\epsilon}, \{(x \geq y)^{\mathtt{t}}, (max(x,y) \doteq y)^{\mathtt{f}}\}_{\epsilon}.$$

- Naming results in

$$\{(max(x,y) \doteq z)^{\mathtt{t}}, P(z,x,y)^{\mathtt{f}}\}_{\epsilon},$$
$$\{(x \geq y)^{\mathtt{f}}, P(x,x,y)^{\mathtt{t}}\}_{\epsilon}, \{(x \geq y)^{\mathtt{t}}, P(y,x,y)^{\mathtt{t}}\}_{\epsilon},$$

  where z is a fresh variable of the sort $\mathbb{Z}$ and P is a fresh predicate symbol of the sort $\mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$.

Finally, VCNF converts signed formulas to literals, and we obtain

- $\{x \not\geq y, max(x,y) \doteq x\}, \{x \geq y, max(x,y) \not\doteq y\}$ in case of expanding;
- $\{max(x,y) \doteq z, \neg P(z,x,y)\}, \{x \not\geq y, P(x,x,y)\}, \{x \geq y, P(y,x,y)\}$ in case of naming.

FOOL2FOL translates (3.1) to $(\forall x : \mathbb{Z})(\forall y : \mathbb{Z})(max(x,y) \doteq g(x,y))$, where g is a fresh function symbol defined by the following formulas:

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

This translation ultimately yields the set of three clauses with two new equalities

$$\{max(x,y) \doteq g(x,y)\}, \{x \not\geq y, g(x,y) \doteq x\}, \{x \geq y, g(x,y) \doteq y\}. \quad \square$$

Both excessive expanding and excessive naming can result in a big CNF. Expanding `if-then-else` expressions in $\mathrm{VCNF}_{\mathrm{FOOL}}$ doubles the number of sequents with occurrences of L, but does not introduce fresh symbols. Naming, on the other hand, adds exactly two new sequents, but introduces a fresh symbol. Both expanding and naming duplicate the condition of the `if-then-else` expression. As discussed previously, $\mathrm{VCNF}_{\mathrm{FOOL}}$ keeps track of the number of occurrences of this condition and names it if this number exceeds the naming threshold. At the same time, expanding constructs two new literals that cannot be named because they might be syntactically distinct and $\mathrm{VCNF}_{\mathrm{FOOL}}$ does not count

occurrences of literals. If the constructed literals contain more `if-then-else` expressions inside, rewriting them might cause exponential increase of the number of sequents.

To balance between these two strategies, we introduce a parameter to $\mathrm{VCNF}_{\mathrm{FOOL}}$ called the `if-then-else` expansion threshold. By default, we heuristically set the `if-then-else` expansion threshold of $\mathrm{VCNF}_{\mathrm{FOOL}}$ to $\log_2 n$, where n is the naming threshold of VCNF. The `if-then-else` expansion threshold of $\mathrm{VCNF}_{\mathrm{FOOL}}$ 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, $\mathrm{VCNF}_{\mathrm{FOOL}}$ names the remaining `if-then-else` expressions.

Similarly to the naming threshold inherited from VCNF, the expansion threshold provides a trade-off between the increase of the number of sequents and the number of introduced symbols. For a large number of `if-then-else` expressions it avoids the exponential increase in the number of sequents. For a small number of `if-then-else` expressions inside an atomic formula it avoids growing the signature.

To compare to FOOL2FOL, we recall that FOOL2FOL 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 an equality. Unlike FOOL2FOL, our new $\mathrm{VCNF}_{\mathrm{FOOL}}$ algorithm avoids introducing new equalities and uses predicate guards for naming, thus avoiding possible self-paramodulation triggered by equality literals.

`if-then-else` Expressions as Formulas

Suppose that ψ is of the form `if` χ `then` γ_1 `else` γ_2. Then, $\mathrm{VCNF}_{\mathrm{FOOL}}$ adds two sequents to C obtained from D by replacing the occurrence of $\psi^\star$ with χ^{f}, ${\gamma_1}^\star$ and χ^{t}, ${\gamma_2}^\star$, respectively.

If done unconditionally, the translation of nested `if-then-else` expressions could lead to an exponential increase in the number of sequents, as the condition formula χ is being copied. However, $\mathrm{VCNF}_{\mathrm{FOOL}}$ inherits from VCNF the mechanism for naming subformulas with many occurrences (as explained in the previous section) which prevents such blowup.

Example. Consider the following property of the *max* function

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

To process (3.2), $\text{VCNF}_{\text{FOOL}}$ first applies twice the rule for existential quantifier inherited from VCNF, obtaining the sequent

$$\{(\texttt{if } max(x,y) \doteq x \texttt{ then } x \geq y \texttt{ else } y \geq x)^{\texttt{t}}\}_{\epsilon}.$$

Then, $\text{VCNF}_{\text{FOOL}}$ applies the rule for the `if-then-else` expression:

$$\{(max(x,y) \doteq x)^{\texttt{f}}, (x \geq y)^{\texttt{t}}\}_{\epsilon}, \{(max(x,y) \doteq x)^{\texttt{t}}, (x \geq y)^{\texttt{f}}\}_{\epsilon}.$$

Finally, $\text{VCNF}_{\text{FOOL}}$ converts signed formulas to literals and obtains the resulting set of clauses

$$\{max(x,y) \not\doteq x, x \geq y\}, \{max(x,y) \doteq x, y \not\geq y\}.$$

In contrast, FOOL2FOL introduces a name for the `if-then-else` expression and translates (3.2) to $(\forall x : \mathbb{Z})(\forall y : \mathbb{Z})P(x,y)$, where P is a fresh predicate symbol of the sort $\mathbb{Z} \times \mathbb{Z}$ with the following definitions:

1. $(\forall x : \mathbb{Z})(\forall y : \mathbb{Z})(max(x,y) \doteq x \Rightarrow P(x,y) \Leftrightarrow x \geq y)$;
2. $(\forall x : \mathbb{Z})(\forall y : \mathbb{Z})(max(x,y) \not\doteq x \Rightarrow P(x,y) \Leftrightarrow y \geq x)$.

These three formulas ultimately yield the following set of clauses:

$\{P(x,y)\}$,
$\{max(x,y) \not\doteq x, \neg P(x,y), x \geq y\}, \{max(x,y) \not\doteq x, P(x,y), x \not\geq y\}$,
$\{max(x,y) \doteq y, \neg P(x,y), y \geq x\}, \{max(x,y) \doteq y, P(x,y), y \not\geq x\}$. □

`let-in` Expressions

Suppose that ψ is $\texttt{let } f(x_1 : \tau_1, \ldots, x_n : \tau_n) = t \texttt{ in } \gamma$. The $\text{VCNF}_{\text{FOOL}}$ algorithms translates ψ either by inlining or by naming, as discussed below. The choice of inlining or naming of `let-in` expressions in the problem is determined by a pre-specified boolean parameter of the algorithm.

Inlining. $\text{VCNF}_{\text{FOOL}}$ adds a sequent to C that is obtained from D by replacing the occurrence of $\psi^\star$ with $\gamma'^\star$. γ' is obtained from γ by replacing each application $f(s_1, \ldots, s_n)$ of an occurrence of f in γ with t' and renaming of binding variables. t' is obtained from t by replacing each free occurrence of $x_1, \ldots, x_n$ in t with $s_1, \ldots, s_n$, respectively. We point out that inlining predicate symbols of zero arity does not hinder identification of tautologies thanks to tautology removal inside sequents.

Naming. $\text{VCNF}_{\text{FOOL}}$ adds a sequent to C that is obtained from D by replacing the occurrence of $\psi^\star$ with $\gamma^\star$. Further, $\text{VCNF}_{\text{FOOL}}$ also adds the sequent $\{f(x_1, \ldots, x_n) \doteq t\}_\epsilon$ to C.

Naming introduces a fresh function or predicate symbol and does not multiply the number of resulting clauses. Inlining, on the other hand, does not introduce any symbols, but can drastically increase the number of clauses. Either of the translations 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 non-boolean 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.

`let-in` Expressions with Tuple Definitions

Suppose that ψ is $\texttt{let}\ (c_1, \ldots, c_n) = s\ \texttt{in}\ \gamma$ where $n > 1$. Let $\tau_1, \ldots, \tau_n$ be the sorts of $c_1, \ldots, c_n$, respectively. Then, the $\text{VCNF}_{\text{FOOL}}$ algorithm

1. introduces a fresh sort τ, a fresh function symbol t of the sort τ, a fresh function symbol g of the sort $\tau_1 \times \ldots \times \tau_n \to \tau$, and n fresh function symbols $\pi_1, \ldots, \pi_n$ (called projection functions), where for each $1 \leq i \leq n$, π_i is of the sort $\tau \to \tau_i$;

2. adds a sequent to C that is obtained from D by replacing every occurrence of $\psi^\star$ with $(\texttt{let}\ t = s'\ \texttt{in}\ \gamma')^\star$. γ' is obtained from γ by replacing each free occurrence of c_i with $\pi_i(t)$ for each $1 \leq i \leq n$. s' is obtained from s by replacing every tuple expression $(s_1, \ldots, s_n)$ with $g(s_1, \ldots, s_n)$;

3. adds sequents to C that axiomatise functions $g, \pi_1, \ldots, \pi_n$. In particular, these state that $\pi_i(g(s_1, \ldots, s_n)) \doteq s_i$ for every $i = 1, \ldots, n$ and that $t_1 \doteq t_2 \Leftrightarrow \bigwedge_{i=1}^{n} \pi_i(t_1) \doteq \pi_i(t_2)$.

Example. Consider a formula that uses a tuple `let-in` expression to swap two constants x and y of the sort $\mathbb{Z}$ before applying a predicate P of the sort $\mathbb{Z} \times \mathbb{Z}$ to them:

$$\texttt{let}\ (x, y) = (y, x)\ \texttt{in}\ P(x, y).$$

To clausify this formula, $\text{VCNF}_{\text{FOOL}}$ firstly converts it to the formula

$$\texttt{let}\ t = g(y, x)\ \texttt{in}\ P(\pi_1(t), \pi_2(t)),$$

where t is a fresh term of the fresh sort τ, and g is a fresh function symbol of the sort $\mathbb{Z} \times \mathbb{Z} \to \tau$, and π_1 and π_2 are projection functions with appropriate axiomatisation. Then, depending on whether inlining or naming is enabled, $\text{VCNF}_{\text{FOOL}}$ result with clauses

$$\{P(\pi_1(g(y, x)), \pi_2(g(y, x)))\} \text{ or } \{P(\pi_1(t'), \pi_2(t'))\}, \{t' \doteq g(y, x)\}$$

respectively, where t' is a fresh constant symbol of the sort τ. □

FOOL2FOL, as described in [50], cannot handle `let-in` expression with tuple definitions.

3.4 Experimental Results

We extended Vampire's VCNF clausification algorithm for standard FOL with our $\text{VCNF}_{\text{FOOL}}$ clausification algorithm for FOOL formulas. The implementation of $\text{VCNF}_{\text{FOOL}}$ comprised about 500 lines of C++ code. Our implementation, benchmarks and results are available at `www.cse.chalmers.se/~evgenyk/fool-cnf-experiments/`.

In what follows, we report on our experimental results obtained by running Vampire on FOOL problems. Whenever we refer to Vampire, we mean the Vampire version extended with our new $\text{VCNF}_{\text{FOOL}}$ clausification algorithm for FOOL. We will write Vampire* for the previous version of Vampire with the FOOL2FOL algorithm of [48]; Vampire* translates FOOL formulas to FOL (after which they are clausified in a standard way) and uses a special inference rule to avoid FOOL self-paramodulation.

For our experiments, we used three sets of benchmarks: (i) problems taken from [70] on reasoning about (co)algebraic datatypes (see Sect. 3.4.1), (ii) examples with both quantifiers and uninterpreted functions taken from the SMT-LIB library [10] (see Sect. 3.4.2), and (iii) benchmarks on proving the partial correctness of loop-free programs (see Sect. 3.4.3). The last benchmark suite is constructed by us to illustrate the use of FOOL in program analysis and verification. As Vampire is the only automated first-order theorem prover supporting FOOL, and in particular `if-then-else` and `let-in` expressions, we could not compare Vampire with any other first-order prover. Further, Vampire* did not yet support tuple expressions in FOOL. Tuple expressions are also not supported by state-of-the-art SMT solvers. For these reasons, we compared

Table 3.1. Runtimes in seconds of provers on the set of 57 unsatisfiable algebraic datatypes problems.

Prover	Solved	Time on solved problems
Vampire	56	23.470
Vampire ⋆	56	31.121
Z3	53	3.615
CVC4	53	25.480

Vampire against Vampire ⋆ and the SMT solvers CVC4 [8] and Z3 [27] only on the experiments from Sect. 3.4.1–3.4.2.

3.4.1 Experiments with Algebraic Datatypes Problems

We used 152 problems about (co)algebraic datatypes taken from [70]. These examples were generated by Isabelle and translated by us to the TPTP syntax [79]. These examples are expressed in FOOL, as they use 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 evaluated the performance of Vampire, Vampire ⋆, CVC4 and Z3 on the unsatisfiable problems in this set. In order to filter out satisfiable problems, we run all the provers on all the problems and only recorded the runs where at least one of the provers reported unsatisfiability. That gave us 57 problems.

We ran both Vampire and Vampire ⋆ with the option `--mode casc`. For the runs of Vampire, the naming threshold was set to 8. We run CVC4 and Z3 with their default options.

Table 3.1 summarises our results. They were obtained on a MacBook Pro with a 2,9 GHz Intel Core i5 and 8 Gb RAM, with a 60 seconds time limit for each benchmark. Vampire and Vampire ⋆ solved the largest number of problems, both provers solved the same problems. 51 problems were solved by all provers. Both Vampire and Vampire ⋆ solved 3 problems, not solved by either CVC4 or Z3. CVC4 and Z3 solved one problem, not solved by either Vampire or Vampire ⋆. Compared to Vampire ⋆, Vampire showed significantly smaller runtime. We therefore conclude that our clausification algorithm for FOOL improved the performance of Vampire on this set of problems.

Table 3.2. Runtimes in seconds of provers on the set of 2191 unsatisfiable SMT-LIB problems.

Prover	Solved	Uniquely solved	Total time on solved
CVC4	2084	55	26,309.47
Vampire	2076	12	22,920.50
Vampire ⋆	1984	9	19,911.69
Z3	1729	4	18,102.96

3.4.2 Experiments with SMT-LIB Problems

As explained in more detail later on (see Section 3.5), FOOL can be regarded as a superset of the SMT-LIB core logic. A theorem prover that supports FOOL can be straightforwardly extended to read problems written in the SMT-LIB syntax. For our experiments using SMT-LIB problems, we used problems in quantified predicate logic with uninterpreted functions stored in the UF subspace of SMT-LIB. These problems 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. The problems taken from SMT-LIB are written in the SMT-LIB 2 syntax. In order to read these problems, we implemented a parser for a sufficient subset of the SMT-LIB 2 language in Vampire. The implementation of the parser comprised about 2,500 lines of C++ code.

We evaluated the 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 solvers proved it to be satisfiable or unsatisfiable, respectively. Otherwise, a problem is marked as `unknown`. In order to filter out satisfiable problems, we ran 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 2191 problems.

We ran Vampire twice on each problem: once with naming of `let-in` expressions and once with inlining (see Sect. 3.3.3). For each run the naming threshold was set to 8. In both runs we also used the option `--mode casc`. For each problem, we recorded the fastest successful run of Vampire. We ran Vampire ⋆ once on each problem with the option `--mode casc`.

Table 3.2 summarises the results of our experiments on the SMT-LIB problems. These results were obtained on the StarExec compute cluster [78] using the time limit of 5 minutes per problem. CVC4 solved the largest number of problems, Vampire solved significantly more than Vampire$\star$, and Z3 solved the least number of problems. None of the provers solved a superset of problems solved by another prover. The "Uniquely solved" column of Table 3.2 shows the number of problems that were solved by each of the provers, but not any of the other ones. 1675 problems were solved by all of the provers, and 2190 problems were solved by at least one of the provers. Vampire solved 111 problems not solved by Vampire$\star$, and Vampire$\star$ solved 19 problems not solved by Vampire.

We also recorded how different translations of `let-in` affected the performance of Vampire. Vampire with inlining of `let-in` expressions solved 61 problems not solved by Vampire without inlining of `let-in` expressions. Vampire without inlining of `let-in` expressions solved 45 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$\star$, we explain it by the fact that Vampire$\star$ always names `if-then-else` expressions, which turns out to be important for solving some problems. Both inlining and naming of `let-in` expressions can make a prover inefficient.

3.4.3 Experiments with FOOL Reasoning about Programs

In this experiment we evaluated Vampire on FOOL problems that express partial correctness property of imperative programs. We obtained these problems manually from a collection of loop-free programs that we in turn generated from a small set of programs with loops by unrolling their loops. Both the benchmarks and the results are available at `www.cse.chalmers.se/~evgenyk/fool-tuple-experiments/`.

We used five small programs with loops annotated with a safety property using the `assert` command. They are listed in Appendix 3.A. Each program contains one loop with one or more `if-then-else` expressions, assignments and tests over integers, integer arrays and booleans. Table 3.3 summarises the programs used in our experiments: the programs `count_two`, `count_two_flag` and `count_three` implement versions of counting elements in an input array using different criteria and ensure that the sum of counted elements equal to the array length; `two_arrays`

and three_arrays sort and compare two, and respectively three arrays element-wise. We unrolled these program loops 2, 3, 4 and 5 times, resulting in a set of 20 annotated loop-free programs. Figure 3.1 shows the program count_two and the program obtained by unrolling three times the loop of count_two.

```
int a[];
int x = 0, y = 0;
for (int i = 0; i < n; i++) {
  if (a[i] > 0) x++; else y++;
}
assert(x + y == n);
```

```
int a[];
int x = 0, y = 0;
if (a[0] > 0) x++; else y++;
if (a[1] > 0) x++; else y++;
if (a[2] > 0) x++; else y++;
assert(x + y == 3);
```

Figure 3.1. The count_two program and the program obtained by unrolling it three times.

For each one of the 20 loop-free benchmarks, we expressed its partial correctness as a TPTP problem using FOOL in the combination of the theory of linear integer arithmetic and the polymorphic theory of arrays [48]. To this end, (i) we formulated the safety assertion as a TPTP conjecture and (ii) expressed the transition relation of the program as a FOOL formula with tuple expressions and let-in expressions with tuple definitions. We refer to [48] for the details of the translation of a program's transition relation to FOOL. In particular, the correctness of this translation is stated in Theorem 1 of that work. Each FOOL formula produces by the translation is linear in the size of the program. Figure 3.2 shows the TPTP translation of the safety property of the count_two_tptp program. It uses the thf subset of the TPTP language, which is the standard subset that contains features of FOOL.

The results of the experiments are summarised in Table 3.3. 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. The first column of the table lists the names of the programs with loops, and columns 2–5 indicate how many time the program loop was unrolled and gives the time needed by Vampire to prove the correctness of the corresponding loop-free program.

Based on the results of this experiment we conclude that Vampire can be used for verification of bounded safety properties of imperative programs.

```
thf(a, type, a: $array($int, $int)).
thf(x, type, x: $int).
thf(y, type, y: $int).

thf(count_two, conjecture,
    $let(x := 0,
    $let(y := 0,
    $let([x, y] := $ite($greater($select(a, 0), 0),
                        $let(x := $sum(x, 1), [x, y]),
                        $let(y := $sum(y, 1), [x, y])),
    $let([x, y] := $ite($greater($select(a, 1), 0),
                        $let(x := $sum(x, 1), [x, y]),
                        $let(y := $sum(y, 1), [x, y])),
    $let([x, y] := $ite($greater($select(a, 2), 0),
                        $let(x := $sum(x, 1), [x, y]),
                        $let(y := $sum(y, 1), [x, y])),
         $sum(x, y) = 3))))))).
```

Figure 3.2. A FOOL translation of the unrolled program in Figure 3.1 written in the TPTP language.

3.5 Related Work

FOOL is a relatively new extension of FOL. We are not aware of any work that explicitly deals with clausifying formulas in this logic. However, connections can be found in work focusing on related fragments and extensions.

Most notably, Wisniewski et al. propose in [96] methods for normalising formulas in higher-order logic (HOL). Similarly to FOOL, HOL natively contains the boolean sort. Wisniewski et al. deal with formulas occurring at argument positions by a technique called *argument extraction* which, similarly to our naming schemes, extends the signature and defines a new symbol outside the original formula. Moreover, also Wisniewski et al. introduce skolem predicates instead of skolem functions when dealing with existential boolean quantifiers. This happens implicitly for them, since in HOL there is no distinction between formulas and terms.

FOOL can be regarded as a superset of SMT-LIB [9] core logic and formulas 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, arity.

Table 3.3. Runtimes in seconds of Vampire on 20 problems encoding partial program correctness.

Problem	2	3	4	5
`count_two`	0.011	0.016	0.030	0.053
`count_two_flag`	0.011	0.017	0.028	0.041
`count_three`	0.023	0.042	0.128	0.522
`two_arrays`	0.026	0.091	0.237	0.263
`three_arrays`	0.446	5.368	8.719	14.886

Despite the similarity of the languages, the technology used by modern SMT solvers [62] differs greatly from that of first-order theorem provers and so do the approaches to normalising the input formula. In particular, as SMT solvers pass the propositional abstraction of the input formula to an efficient SAT solver there is no great need to optimise extensions of the signature and clausification usually follows the simple Tseitin encoding [91] of the formula tree. Moreover, modern SMT solvers employ an alternative approach to dealing with quantifiers over interpreted sorts such as the booleans, which is complementary to skolemisation and relies on a guidance by counter-examples [71] or on model-based projections [15].

Finally, it is interesting to note that our $\text{VCNF}_{\text{FOOL}}$ algorithm naturally translates a quantified boolean formula (QBF), as realised in the FOOL language, into a CNF in effectively propositional logic (EPR). Specifically, every literal in this translation is a skolem predicate applied to boolean variables and constants *true* and *false*. This result is similar to the one proposed in [76], where the authors explicitly focus on QBF as the source and EPR as the target language, respectively. Obtaining a formula in EPR is a desirable property since there are first-order proving methods known to be efficient for dealing with the fragment (see e.g. [47]).

3.6 Conclusion and Future Work

Applications of program analysis and verification rely on SAT/SMT solvers and/or theorem provers to reason about program properties formulated in various logics. The efficiency of SAT/SMT solvers and theorem provers critically depends on the used clausification algorithm. In this paper we presented a new clausification algorithm, called $\text{VCNF}_{\text{FOOL}}$, for formulas expressed in FOOL. Our algorithm is a non-trivial extension of the recent VCNF clausification algorithm for standard first-order logic. $\text{VCNF}_{\text{FOOL}}$ for FOOL introduces skolem predicates over boolean vari-

ables, avoids equalities over boolean variables, and uses formula naming and tautology elimination on complex formulas. It also avoids excessively duplicating clauses and introducing too many new symbols. Thanks to the our new $\text{VCNF}_{\text{FOOL}}$ algorithm, proving FOOL formulas requires neither an axiomatisation of the boolean sort nor modifications in superposition calculus. We implemented our work in Vampire and experimentally showed its benefits on a large number of examples. For future work we are interested in developing further criteria for controlling naming and inlining expressions during clausification. Using FOOL for more complex applications of program analysis is another interesting venue to exploit.

Acknowledgments

We thank Andrew Reynolds for an explanation on how state-of-the-art SMT solvers deal with clausification and quantifiers.

This work has been supported by the ERC Starting Grant 2014 SYMCAR 639270, the Wallenberg Academy Fellowship 2014, the Swedish VR grant D0497701 and the Austrian research project FWF S11409-N23.

Appendix 3.A. Imperative Programs with Loops and `if-then-else`

```
int a[];
int x = 0, y = 0;
for (int i = 0; i < n; i++)
{
  if (a[i] > 0) x++;
  else y++;
}
assert(x + y == n);
```

count_two

```
int a[];
int x = 0, y = 0, z = 0;
for (int i = 0; i < n; i++)
{
  if (a[i] < 0) x++;
  else {
    if (a[i] > 5) y++;
    else z++;
  }
}
assert(x + y + z == n);
```

count_three

```
int a[], b[];
for (int i = 0; i < n; i++)
{
  if (a[i] > b[i]) {
    int t = a[i];
    a[i] = b[i];
    b[i] = t;
  }
}
for (int i = 0; i < n; i++)
{
  assert(a[i] <= b[i]);
}
```

two_arrays

```
int a[];
bool b;
int x = 0, y = 0;
for (int i = 0; i < n; i++)
{
  b = a[i] > 0;
  if (b) x++; else y++;
}
assert(x + y == n);
```

count_two_flag

```
int a[], b[], c[];
for (int i = 0; i < n; i++)
{
  if (a[i] > b[i]) {
    int t = a[i];
    a[i] = b[i];
    b[i] = t;
  }
  if (b[i] > c[i]) {
    int t = b[i];
    b[i] = c[i];
    c[i] = t;

    if (a[i] > b[i]) {
      t = a[i];
      a[i] = b[i];
      b[i] = t;
    }
  }
}
for (int i = 0; i < n; i++)
{
  assert(a[i] <= b[i]);
  assert(b[i] <= c[i]);
}
```

three_arrays

Chapter 4

A FOOLish Encoding of the Next State Relations of Imperative Programs

Evgenii Kotelnikov, Laura Kovács and Andrei Voronkov

Abstract. Automated theorem provers are routinely used in program analysis and verification for checking program properties. These properties are translated from program fragments to formulas expressed in the logic supported by the theorem prover. Such translations can be complex and require deep knowledge of how theorem provers work in order for the prover to succeed on the translated formulas. Our previous work introduced FOOL, a modification of first-order logic that extends it with syntactical constructs resembling features of programming languages. One can express program properties directly in FOOL and leave translations to plain first-order logic to the theorem prover. In this paper we present a FOOL encoding of the next state relations of imperative programs. Based on this encoding we implement a translation of imperative programs annotated with their pre- and post-conditions to partial correctness properties of these programs. We present experimental results that demonstrate that program properties translated using our method can be efficiently checked by the first-order theorem prover Vampire.

Published in the *Proceedings of the 9th International Joint Conference on Automated Reasoning*, pages 405–421. Springer, 2018.

4.1 Introduction

Automated program analysis and verification requires discovering and proving program properties ensuring program correctness. These program properties are usually expressed in combined theories of various data structures, such as integers and arrays. SMT solvers and first-order theorem provers that are used to check these properties need efficient handling of both theories and quantifiers. Moreover, formalisation of the program properties in the logic supported by the SMT solver or theorem prover plays a crucial role in making the prover succeed proving program correctness.

The translation of program properties into logical formulas accepted by a theorem prover is not straightforward. The reason for this is 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 needs to implement a translation of these properties to the language of the theorem prover. Such translations can be complex and error prone. Furthermore, one might need deep knowledge of how theorem provers work to obtain formulas in a form that theorem provers can handle efficiently.

Program verification systems reduce the mismatch between program properties and their formalisation as logical formulas from two ends. On the one hand, intermediate verification languages, such as Boogie [57] and WhyML [31], are designed to represent programs and their properties in a way that is friendly for translations to logic. On the other hand, theorem provers extend their supported logics with syntactic constructs that mirror those of programming languages.

Our previous work introduced FOOL [50], a modification of many-sorted first-order logic (FOL). FOOL extends FOL with syntactical constructs such as `if-then-else` and `let-in` expressions. These constructs can be used to naturally express program properties about conditional statements and variable updates. Users of a theorem prover that supports FOOL do not need to invent translations for these features of programming languages and can use features of FOOL directly. It allows the theorem prover to apply its own translation to FOL that it can use efficiently. We extended the Vampire theorem prover [54] to support FOOL [48] and designed an efficient clausification algorithm VCNF [49] for FOOL.

In summary, FOOL extends FOL with the following constructs.

- First-class boolean sort — one can define function and predicate symbols with boolean arguments and use quantifiers over the boolean sort.
- Boolean variables used as formulas.
- Formulas used as arguments to function and predicate symbols.
- Expressions of the form `if` φ `then` s `else` t, where φ is a formula, and s and t are either both terms or formulas.
- Expressions of the form `let` $D_1; \ldots; D_k$ `in` t, where $k > 0$, t is either a term or a formula, and $D_1, \ldots, D_k$ are simultaneous definitions, each of the form
  1. $f(x_1 : \sigma_1, \ldots, x_n : \sigma_n) = s$, where $n \geq 0$, f can be a function or a predicate symbol, and s is either a term or a formula;
  2. $(c_1, \ldots, c_n) = s$, where $n > 1$, $c_1, \ldots, c_n$ are constant symbols of the sorts $\sigma_1, \ldots, \sigma_n$, respectively, and s is a tuple expression. A tuple expression is inductively defined to be either
     (a) $(s_1, \ldots, s_n)$, where $s_1, \ldots, s_n$ are terms of the sorts $\sigma_1, \ldots, \sigma_n$, respectively;
     (b) `if` φ `then` s_1 `else` s_2, where φ is a formula, and s_1 and s_2 are tuple expressions; or
     (c) `let` $D_1; \ldots; D_k$ `in` s', where $D_1; \ldots; D_k$ are definitions, and s' is a tuple expression.

To our knowledge, no other logic, efficiently implemented in automated theorem provers, contains these constructs. Some constructs of FOOL have been previously implemented in interactive and higher-order theorem provers. However, there was no special emphasis on the efficiency or friendliness of the translation for the following processing by automatic provers.

In this paper, we extend our previous work on FOOL by demonstrating the efficient use of FOOL for program analysis. To this end, we give an efficient encoding of the next state relations of imperative programs in FOOL. Let us motivate our work with the simple program on Figure 4.1. This program contains an `if` statement and assignments to integer variables. The `assert` statement ensures that `x` is never greater than `y` after execution of the `if` statement.

To check that the given program assertion holds using an automated theorem prover, one has to express this assertion as a logical formula.

```
if (x > y) {
  t := x;
  x := y;
  y := t;
}
assert(x <= y);
```

Figure 4.1. An imperative program with an `if` statement.

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

Figure 4.2. A FOOL encoding of the program assertion on Figure 4.1.

For that, one has to express the updated values of x and y after the sequence of assignments. For example, one can compute the updated value of each individual variable separately for each possible execution trace. However, this approach suffers from a bloated resulting formula that will contain duplicating parts of the program. A more common technique is to first convert a program to a static single assignment (SSA) form. This conversion introduces a new intermediate variable for each assignment and creates a smaller translated formula.

Both excessive naming and excessive duplication of program expressions can make the resulting logical formula very hard for a first-order theorem prover. The encoding of the next state relations of imperative programs given in this paper avoids both by using a FOOL formula that closely matches the structure of the original program (Section 4.3). This way the decision between introducing new symbols and duplicating program expressions is left to the theorem prover that is better equipped to make it. The assertion of the program in Figure 4.1 is concisely expressed with our encoding as the FOOL formula on Figure 4.2.

While FOOL offers a concise representation of some programming constructs, the efficient implementation of FOOL poses a challenge for first-order theorem provers since their performance on various translations to CNF can be hampered by the (unintended) use of constructs interfering with their internal implementation, including the use of orderings, selection and the saturation algorithm. For example, to deal with the boolean sort, it is not uncommon to add an axiom like $(\forall x)(x = 0 \vee x = 1)$ for this sort. Even this simple axiom can cause a considerable growth of the search space, especially when used with certain term orderings. To address the challenge of dealing with full FOOL, one needs experimental

comparison of various translations or various implementations of FOOL. Our paper is the first one to make such an experimental comparison.

Our encoding uses tuple expressions and `let-in` expressions with tuple definitions, available in FOOL. We extend and generalise the use of tuples in first-order theorem provers by introducing a polymorphic theory of first class tuples (Section 4.2). In this theory one can define tuple sorts and use tuples as terms.

Our encoding can be efficiently used in automated program analysis and verification. To demonstrate this, we report on our experimental results obtained by running Vampire on program verification problems (Section 4.4). These verification problems are partial correctness properties that we generated from a collection of imperative programs using an implementation of our encoding to FOOL as well as other tools.

Contributions. We summarise the main contributions of this paper below.

1. We define an encoding of the next state relation of imperative programs in FOOL and show that it is sound (Section 4.3). Using this encoding, we define a translation of certain properties of imperative programs to FOOL formulas.

2. We present a polymorphic theory of first class tuples and its implementation in Vampire (Section 4.2). To our knowledge, Vampire is the only superposition-based theorem prover to support this theory.

3. We present experimental results obtained by running Vampire on a collection of benchmarks expressing partial correctness properties of imperative programs (Section 4.4). We generated these benchmarks using an implementation of our encoding to FOOL and other tools. Our results show Vampire is more efficient on the FOOL encoding of partial correctness properties, compared with other translations.

4.2 Polymorphic Theory of First Class Tuples

The use of tuple expressions in FOOL is limited. They can only occur on the right hand side of a tuple definition in `let-in`. One cannot use a tuple expression elsewhere, for example, as an argument to a function or predicate symbol.

In this section we describe the theory of first class tuples that enables a more generic use of tuples. This theory contains tuple sorts and tuple terms. Both of them are first class — one can define function and predicate

symbols with tuple arguments, quantify over the tuple sort, and use tuple terms as arguments to function and predicate symbols. Tuple expressions in FOOL, combined with the polymorphic theory of tuples, are tuple terms.

Definition. The polymorphic theory of tuples is the union of theories of tuples parametrised by tuple arity $n > 0$ and sorts $\tau_1, \ldots, \tau_n$.

A theory of first class tuples is a first-order theory that contains a sort $(\tau_1, \ldots, \tau_n)$, function symbols $t : \tau_1 \times \ldots \times \tau_n \to (\tau_1, \ldots, \tau_n)$, $\pi_1 : (\tau_1, \ldots, \tau_n) \to \tau_1, \ldots, \pi_n : (\tau_1, \ldots, \tau_n) \to \tau_n$, and two axioms. The function symbol t constructs a tuple from given terms, and function symbols $\pi_1, \ldots, \pi_n$ project a tuple to its individual elements. For simplicity we will write $(t_1, \ldots, t_n)$ instead of $t(t_1, \ldots, t_n)$ to mean a tuple of terms $t_1, \ldots, t_n$. The tuple axioms are

1. exhaustiveness

$$(\forall x_1 : \tau_1) \ldots (\forall x_n : \tau_n) \\ (\pi_1((x_1, \ldots, x_n)) \doteq x_1 \wedge \ldots \wedge \pi_n((x_1, \ldots, x_n)) \doteq x_n);$$

2. injectivity

$$(\forall x_1 : \tau_1) \ldots (\forall x_n : \tau_n)(\forall y_1 : \tau_1) \ldots (\forall y_n : \tau_n) \\ ((x_1, \ldots, x_n) \doteq (y_1, \ldots, y_n) \Rightarrow x_1 \doteq y_1 \wedge \ldots \wedge x_n \doteq y_n).$$

Tuples are ubiquitous in mathematics and programming languages. For example, one can use the tuple sort $(\mathbb{R}, \mathbb{R})$ as the sort of complex numbers. Thus, the term (a, b), where $a : \mathbb{R}$ and $b : \mathbb{R}$ represents a complex number $a + bi$. One can define the addition function $plus : (\mathbb{R}, \mathbb{R}) \times (\mathbb{R}, \mathbb{R}) \to (\mathbb{R}, \mathbb{R})$ for complex numbers with the formula

$$(\forall x : (\mathbb{R}, \mathbb{R}))(\forall y : (\mathbb{R}, \mathbb{R})) \\ (plus(x, y) \doteq (\pi_1(x) + \pi_1(y), \pi_2(x) + \pi_2(y))), \tag{4.1}$$

where $+$ denotes addition for real numbers.

Tuple terms can be used as tuple expressions in FOOL. If $(c_1, \ldots, c_n) = s$ is a tuple definition inside a `let-in`, where $c_1, \ldots, c_n$ are constant symbols of sorts $\tau_1, \ldots, \tau_n$, respectively, then tuple expression s is a term of the sort $(\tau_1, \ldots, \tau_n)$.

It is not hard to extend tuple definitions to allow arbitrary tuple terms of the correct sort on the right hand side of $=$. For example, one can

use a variable of the tuple sort. With such extension, Formula 4.1 can be equivalently expressed using a `let-in` with two simultaneous tuple definitions as follows

$$\begin{aligned}&(\forall x : (\mathbb{R}, \mathbb{R}))(\forall y : (\mathbb{R}, \mathbb{R}))\\ &\quad (\mathit{plus}(x, y) \doteq \texttt{let } (a, b) = x;\ (c, d) = y \texttt{ in } (a + c, b + d)).\end{aligned} \tag{4.2}$$

Implementation. Vampire implements reasoning with the polymorphic theory of tuples by adding corresponding tuple axioms when the input uses tuple sorts and/or tuple functions. For each tuple sort $(\tau_1, \ldots, \tau_n)$ used in the input, Vampire defines a term algebra [52] with the single constructor t and n destructors $\pi_1, \ldots, \pi_n$. Then Vampire adds the corresponding term algebra axioms, which coincide with the tuple theory axioms.

Vampire reads formulas written in the TPTP language [86]. The TFX subset[1] of TPTP contains a syntax for tuples and `let-in` expressions with tuple definitions. The sort $(\mathbb{R}, \mathbb{R})$ is represented in TFX as `[$real,$real]` and the term $(a + c, b + d)$ is represented in TFX as `[$sum(a,c),$sum(b,d)]`. Formula 4.2 can be expressed in TPTP as

```
tff(plus, type,
    plus: ([$real,$real] * [$real,$real]) > [$real,$real]).
tff(plus_def, axiom,
    ![X: [$real, $real], Y: [$real, $real]]:
      (plus(X, Y) = $let([[a: $real, b: $real],
                          [c: $real, d: $real]],
                         [a, b] := X; [c, d] := Y,
                         [$sum(a, c), $sum(b, d)]))).
```

Vampire translates `let-in` with tuple definitions to clausal normal form of first-order logic using the VCNF clausification algorithm [49].

4.3 Imperative Programs to FOOL

In this section we discuss an efficient translation of imperative programs to FOOL. To formalise the translation we define a restricted imperative programming language and its denotational semantics in Section 4.3.1. This language is capable of expressing variable updates, `if-then-else`, and sequential composition. Then, we define an encoding of the next state relation for programs of this language, and state the soundness property

[1] http://www.cs.miami.edu/~tptp/TPTP/Proposals/TFXTHX.html

of this encoding in Section 4.3.2. Finally, in Section 4.3.3 we show a translation that converts a program, annotated with its pre-conditions and post-conditions, to a FOOL formula that expresses the partial correctness property of that program.

We give (rather standard) definitions of our programming language and its semantics and use them to define the main contributions of this section: the encoding of the next state relation (Definition 4.6) and soundness of this encoding (Theorem 4.1).

4.3.1 An Imperative Programming Language

We define a programming language with assignments to typed variables, `if-then-else`, and sequential composition. We omit variable declarations in our language and instead assume for each program a set of program variables V and a type assignment η. η is a function that maps each program variable into a type. Each type is either `int`, `bool`, or $\texttt{array}(\sigma, \tau)$, where σ and τ are types of array indexes and array values, respectively. In the sequel we will assume that V and η are arbitrary but fixed.

Programs in our language select and update elements of arrays, including multidimensional arrays. We do not introduce a distinguished type for multidimensional arrays but instead use nested arrays. We write $\texttt{array}(\sigma_1, \ldots, \sigma_n, \tau)$, $n > 1$, to mean the nested array type $\texttt{array}(\sigma_1, \texttt{array}(\ldots, \texttt{array}(\sigma_n, \tau)\ldots))$.

Definition 4.1. An *expression* of the type τ is defined inductively as follows.

1. An integer n is an expression of the type `int`.
2. Symbols `true` and `false` are expressions of the type `bool`.
3. If $\eta(x) = \tau$, then x is an expression of the type τ.
4. If $\eta(x) = \texttt{array}(\sigma_1, \ldots, \sigma_n, \tau)$, $n > 0$, $e_1, \ldots, e_n$ are expressions of types $\sigma_1, \ldots, \sigma_n$, respectively, then $x[e_1, \ldots, e_n]$ is an expression of the type τ.
5. If e_1 and e_2 are expressions of the type τ, then $e_1 \doteq e_2$ is an expression of the type `bool`.
6. If e_1 and e_2 are expressions of the type `int`, then $-e_1$, $e_1 + e_2$, $e_1 - e_2$, $e_1 \times e_2$ are expressions of the type `int`.
7. If e_1 and e_2 are expressions of the type `int`, then $e_1 < e_2$ is an expression of the type `bool`.

8. If e_1 and e_2 are expression of the type `bool`, then $\neg e_1$, $e_1 \vee e_2$, $e_1 \wedge e_2$ are expressions of the type `bool`. □

Definition 4.2. A *statement* is defined inductively as follows.

1. `skip` is an empty statement.
2. If $\eta(x_1) = \tau_1, \ldots, \eta(x_n) = \tau_n, n \geq 1$ and $e_1, \ldots, e_n$ are expressions of the types $\tau_1, \ldots, \tau_n$, respectively, then $x_1, \ldots, x_n := e_1, \ldots, e_n$ is a statement.
3. If $\eta(x) = \texttt{array}(\sigma_1, \ldots, \sigma_n, \tau)$, $n \geq 1$, and $e_1, \ldots, e_n, e$ are expressions of types $\sigma_1, \ldots, \sigma_n, \tau$, respectively, then $x[e_1, \ldots, e_n] := e$ is a statement.
4. If e is an expression of the type `bool`, s_1 and s_2 are statements, and at least one of s_1, s_2 is not `skip`, then `if` e `then` s_1 `else` s_2 is a statement.
5. If s_1 and s_2 are statements and neither of them is `skip`, then $s_1 \,;\, s_2$ is a statement. □

We say that $x_1, \ldots, x_n$ in the statement $x_1, \ldots, x_n := e_1, \ldots, e_n$ and x in the statement $x[e_1, \ldots, e_n] := e$ are *assigned program variables*. For each statement s we denote by updates(s) the set of all assigned program variables that occur in s.

We define the semantics of the programming language by an interpretation function $[\![-]\!]$ for types, expressions and statements. The interpretation of a type is a set: $[\![\texttt{int}]\!] = \mathbb{Z}$, $[\![\texttt{bool}]\!] = \{0, 1\}$, and $[\![\texttt{array}(\tau, \sigma)]\!] = [\![\tau]\!] \to [\![\sigma]\!]$. The interpretation of expressions and statements is defined using *program states*, that is, mappings of program variables $x \in V$, $\eta(x) = \tau$ to elements of $[\![\tau]\!]$.

Definition 4.3. Let e be an expression of the type τ. The *interpretation* $[\![e]\!]$ is a mapping from program states to $[\![\tau]\!]$ defined inductively as follows.

1. $[\![n]\!]$ maps each state to n, where n is an integer.
2. $[\![\texttt{true}]\!]$ maps each state to 1.
3. $[\![\texttt{false}]\!]$ maps each state to 0.
4. $[\![x]\!]$ maps each st to $st(x)$.
5. $[\![x[e_1, \ldots, e_n]]\!]$ maps each st to $st(x)([\![e_1]\!](st)) \ldots ([\![e_n]\!](st))$.

6. $[\![e_1 \oplus e_2]\!]$ maps each st to $[\![e_1]\!](st) \oplus [\![e_2]\!](st)$, where $\oplus \in \{\doteq, +, -, \times, <, \vee, \wedge\}$.

7. $[\![\neg e]\!]$ maps each st to $\neg [\![e]\!](st)$. □

Definition 4.4. Let s be a statement. The *interpretation* $[\![s]\!]$ is a mapping between program states defined inductively as follows.

1. $[\![\texttt{skip}]\!]$ is the identity mapping.

2. $[\![x_1, \ldots, x_n \coloneqq e_1, \ldots, e_n]\!]$ maps each st to st' such that $st'(x_i) = [\![e_i]\!](st)$ for each $1 \leq i \leq n$ and otherwise coincides with st.

3. $[\![x[e_1, \ldots, e_n] \coloneqq e]\!]$ maps each st to st' such that

$$st'(x)([\![e_1]\!](st)) \ldots ([\![e_n]\!](st)) = [\![e]\!](st)$$

   and otherwise coincides with st.

4. $[\![\texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2]\!]$ maps each st to $[\![s_1]\!](st)$ if $[\![e]\!](st) = 1$ and to $[\![s_2]\!](st)$ otherwise.

5. $[\![s_1 \,;\, s_2]\!]$ is $[\![s_2]\!] \circ [\![s_1]\!]$. □

4.3.2 Encoding the Next State Relation

Our setting is FOOL extended with the theory of linear integer arithmetic, the polymorphic theory of arrays [48], and the polymorphic theory of first class tuples (Section 4.2). The theory of linear integer arithmetic includes the sort $\mathbb{Z}$, the predicate symbol $<$, and the function symbols $+$, $-$, and $\times$. The theory of arrays includes the sort $\mathit{array}(\tau, \sigma)$ for all sorts τ and σ, and function symbols *select* and *store*. 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. We point out that sorts *bool*, $\mathbb{Z}$, and $\mathit{array}(\sigma, \tau)$ mirror types `bool`, `int` and $\texttt{array}(\sigma, \tau)$ of our programming language, and have the same interpretations.

We represent multidimensional arrays in FOOL as nested arrays[2]. To this end we (i) inductively define $\mathit{select}(a, i_1, \ldots, i_n)$, $n > 1$, to be $\mathit{select}(\mathit{select}(a, i_1), i_2, \ldots, i_n)$; and (ii) inductively define $\mathit{store}(a, i_1, \ldots, i_n, e)$, $n > 1$, to be $\mathit{store}(a, i_1, \mathit{store}(\mathit{select}(a, i_1), i_2, \ldots, i_n, e))$.

[2]Multidimensional arrays can be represented in FOOL also as arrays with tuple indexes. We do not discuss such representation in this work.

Our encoding of the next state relation produces FOOL terms that use program variables as constants and do not use any other uninterpreted function or predicate symbols. In the sequel we will only consider such FOOL terms. For these FOOL terms, η is a type assignment and each program state can be extended to a η-interpretation, the details of this extension are straightforward (we refer to [50] for the semantics of FOOL). We will use program states as η-interpretations for FOOL terms. For example we will write $\text{eval}_{st}(t)$ for the value of t in st, where t is a FOOL term and st is a program state. We will say that a program state st satisfies a FOOL formula φ if $\text{eval}_{st}(\varphi) = 1$.

To define the encoding of the next state relation we first define a translation of expressions to FOOL terms. Our encoding applies this translation to each expression that occurs inside a statement.

Definition 4.5. Let e be an expression of the type τ. $\mathcal{T}(e)$ is a FOOL term of the sort τ, defined inductively as follows.

$$\begin{aligned}
\mathcal{T}(n) &= n, \text{where } n \text{ is an integer.}\\
\mathcal{T}(\texttt{true}) &= \mathit{true}.\\
\mathcal{T}(\texttt{false}) &= \mathit{false}.\\
\mathcal{T}(x) &= x.\\
\mathcal{T}(x[e_1, \ldots, e_n]) &= \mathit{select}(x, \mathcal{T}(e_1), \ldots, \mathcal{T}(e_n)).\\
\mathcal{T}(e_1 \oplus e_2) &= \mathcal{T}(e_1) \oplus \mathcal{T}(e_2), \text{where } \oplus \in \{\doteq, +, -, <, \times, \vee, \wedge\}.\\
\mathcal{T}(-e) &= -\mathcal{T}(e).\\
\mathcal{T}(\neg e) &= \neg\mathcal{T}(e).
\end{aligned}$$

□

Lemma 4.1. $\text{eval}_{st}(\mathcal{T}(e)) = [\![\, e \,]\!]\,(st)$ for each expression e and state st.

□

Proof. By structural induction on e. □

Definition 4.6. Let s be a statement. $\mathcal{N}(s)$ is a mapping between FOOL terms of the same sort, defined inductively as follows.

1. $\mathcal{N}(\texttt{skip})$ is the identity mapping.
2. $\mathcal{N}(x_1, \ldots, x_n \coloneqq e_1, \ldots, e_n)$ maps t to

$$\texttt{let } (x_1, \ldots, x_n) = (\mathcal{T}(e_1), \ldots, \mathcal{T}(e_n)) \texttt{ in } t.$$

3. $\mathcal{N}(x[e_1, \ldots, e_n] \coloneqq e)$ maps t to

$$\texttt{let } x = \mathit{store}(x, \mathcal{T}(e_1), \ldots, \mathcal{T}(e_n), \mathcal{T}(e)) \texttt{ in } t.$$

4. $\mathcal{N}(\texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2)$ maps t to

$$\begin{aligned}\texttt{let } (x_1, \ldots, x_n) = \texttt{ if } \mathcal{T}(e) \texttt{ then } \mathcal{N}(s_1)((x_1, \ldots, x_n)) \\ \texttt{else } \mathcal{N}(s_2)((x_1, \ldots, x_n)) \\ \texttt{in } t,\end{aligned}$$

where $\text{updates}(s_1) \cup \text{updates}(s_2) = \{x_1, \ldots, x_n\}$.

5. $\mathcal{N}(s_1 \,;\, s_2)$ is $\mathcal{N}(s_1) \circ \mathcal{N}(s_2)$. □

The following theorem is the soundness property of translation $\mathcal{N}$. Essentially, it states that $\mathcal{N}$ encodes the semantics of a given statement as a FOOL formula.

Theorem 4.1. $\text{eval}_{st}(\mathcal{N}(s)(t)) = \text{eval}_{[\![\, s \,]\!](st)}(t)$ for each statement s, state st and FOOL term t. □

Proof. By structural induction on s. □

4.3.3 Encoding the Partial Correctness Property

We use the encoding of the next state relation to generate partial correctness properties of programs annotated with their pre-conditions and post-conditions.

We define an *annotated program* to be a Hoare triple $\{\varphi\}\, s\, \{\psi\}$, where s is a statement, and φ and ψ are formulas in first-order logic. We say that $\{\varphi\}\, s\, \{\psi\}$ is correct if for each program state st that satisfies φ, $[\![\, s \,]\!]\,(st)$ satisfies ψ. We translate each annotated program $\{\varphi\}\, s\, \{\psi\}$ to the FOOL formula $\varphi \Rightarrow \mathcal{N}(s)(\psi)$.

Theorem 4.2. Let $\{\varphi\}\, s\, \{\psi\}$ be an annotated program. The FOOL formula $\varphi \Rightarrow \mathcal{N}(s)(\psi)$ is valid iff $\{\varphi\}\, s\, \{\psi\}$ is correct. □

Proof. Directly follows from Theorem 4.1. □

We point out the following two properties of the encoding $\mathcal{N}$. First, the size of the encoded formula is $O(v \cdot n)$, where v is the number of variables in the program and n is the program size as each program statement is used once with one or two instances of $(x_1, \ldots, x_n)$. Second, the encoding does not introduce any new symbols. When we translate program correctness properties to FOL, both an excessive number of new symbols and an excessive size of the translation might make the encoded formula hard for a theorem prover. Instead of balancing between the two, encoding to FOOL leaves the decision to the theorem prover.

4.4 Experiments

In this section we describe our experiments on comparing the performance of the Vampire theorem prover [54] on FOOL and on translations of program properties to FOL. We used a collection of 50 programs written in the Boogie verification language [57]. Each of these programs uses only variable assignments, `if-then-else` statements, and sequential composition and is annotated with its pre-conditions and post-conditions, expressed in first-order logic. From this collection of programs we generated the following three sets of benchmarks.

1. 50 problems in first-order logic written in the SMT-LIB language [10]. We generated these problems by running the front end of the Boogie [6] verifier.
2. 50 FOOL problems with tuples generated by running our implementation of the translation from Section 4.3.3, named Voogie.
3. 50 FOOL problems generated by running the BLT [23] translator.

We point out that in our experiments we do not aim to compare methods of program verification or specific verification tools. Rather, we compare different ways of translating realistic verification problems for theorem provers.

In what follows, we describe the collection of imperative programs used in our experiments (Section 4.4.1) and discuss our set of benchmarks (Section 4.4.2). All properties that we deal with use integers and arrays, as well as universal and existential quantifiers. To verify these properties one has to reason in the combination of theories and quantifiers. We briefly describe how Vampire implements this kind of reasoning in Section 4.4.3. Our experimental results are summarised in Tables 4.1–4.3 and discussed in Section 4.4.4.

4.4.1 Examples of Imperative Programs

We demonstrate the work of our translation on a collection of imperative programs that only use variable assignments, `if-then-else` statements, and sequential composition. Unfortunately, no large collections of such programs are available. There are many benchmarks for software verification tools, but most of them use control flow statements not covered in this work, such as gotos and exceptions. We also cannot use benchmarks from the hardware verification and model checking communities, because

they are mostly about boolean values and bit-vectors. For our experiments we generated our own imperative programs in two steps described below.

First, we crafted 10 programs that implement textbook algorithms and solutions to program verification competitions. Each program uses variables of the integer, boolean, and array type. Each program contains a single `while` loop of the form `while` e `do` s, where e is a boolean expression and s is a statement. In addition, each program contains variable assignments, `if-then-else` statements, and sequential composition. We annotated each program with its pre-condition φ and each loop with its invariant ψ. The formulas φ and ψ are expressed in first-order logic.

Then, we unrolled the loop of each program k times, where k is an integer between 1 and 5. This resulted in 50 loop-free programs that retain the annotated properties. Each program encodes the loop invariant property of the original program. Multiple unrollings provide us with programs with long sequences of variables updates, `if-then-else` statements and compositions, which are convenient for our experiments. Our loop unrolling program transformation consisted of the following steps.

1. Introduce a fresh boolean variable *bad* that encodes the underspecified state of the program.

2. Construct a guarded loop iteration i as

$$\texttt{if } \neg e \texttt{ then } \mathit{bad} \coloneqq \mathit{true} \texttt{ else skip}\,; s.$$

3. Construct a sequence of iterations $i\,; \ldots ; i$, where i is repeated k times.

4. Finally, construct the annotated program

$$\{\varphi \wedge \psi\}\, i\,; \ldots ; i\, \{\neg \mathit{bad} \Rightarrow \psi\}.$$

It is not hard to show that if a program with a loop satisfies its specification, then the Hoare triple resulting in step 4 of the above transformation also holds.

We wrote our example programs with loops as well as their loop-free unrolled versions in the Boogie language. Boogie can unroll loops automatically, but introduces `goto` statements that our translation does not support. For this reason, we used the loop unrolling described above.

An example of our loop unrolling is available at http://www.cse.chalmers.se/~evgenyk/ijcar18/. It shows the maxarray program with a loop from our collection and a program generated from maxarray by unrolling its loop twice.

4.4.2 Benchmarks

We used the 50 annotated loop-free programs and generated their partial correctness statements using Boogie, Voogie and BLT. These statements are encoded as unsatisfiable problems in first-order logic and FOOL. Our collection of imperative programs with loops, their loop-free unrollings and benchmarks expressed in the TPTP language [79] is available at http://www.cse.chalmers.se/~evgenyk/ijcar18/. The TPTP benchmarks are also available, along with other FOOL problems, on the TPTP website http://tptp.org.

The Boogie verifier generates verification conditions as formulas in first-order logic written in the SMT-LIB language and uses the SMT solver Z3 [27] to check these formulas. We ran Boogie with the option /proverLog to print the generated formulas on each of our annotated loop-free programs and in this way obtained a collection of 50 SMT-LIB benchmarks.

Voogie is our implementation of the translation described in Section 4.3. It takes as input programs written in a fragment of the Boogie language and generates FOOL formulas written in the TPTP language. The source code of Voogie is available at https://github.com/aztek/voogie.

The fragment of the Boogie language supported by Voogie can be seen as the smallest fragment that is sufficient to represent the loop-free programs in our collection. This fragment consists of (i) top level variable declarations; (ii) a single procedure main annotated with its pre- and post-conditions; (iii) assignments to variables, including parallel assignments, and assignments to array elements; (iv) if-then-else statements; and (v) arithmetic and boolean operations. Running Voogie on each loop-free program in our collection gave us 50 TPTP benchmarks. An example of the TPTP benchmark obtained from running Voogie on the maxarray program with its loops unrolled twice is available at http://www.cse.chalmers.se/~evgenyk/ijcar18/.

BLT (Boogie Less Triggers) [23] is an automatic tool that takes Boogie programs as input and generates their verification conditions in first-order logic written in the TPTP language. BLT has an experimental feature of generating FOOL formulas with tuple let-in and tuple expressions

to represent next state values of program variables in a style similar to Voogie. At the time of our experiments, this feature was not stable enough, and we did not enable it. Running BLT with its default configuration on each of the 50 loop-free programs in our collection gave us 50 TPTP benchmarks.

The representation of program expressions coincides in all three translations. All translations use the theory of linear integer arithmetic and the theory of arrays as realised in their respective languages.

4.4.3 Theories and Quantifiers in Vampire

Vampire's main algorithm is saturation of a set of first-order clauses using the resolution and superposition calculus. Vampire also implements the AVATAR architecture [93] for splitting clauses. The idea behind AVATAR is to use a SAT or an SMT solver to guide proof search. AVATAR selects sub-problems for the saturation-based prover to tackle by making decisions over a propositional abstraction of the clause search space. The `-sas` option of Vampire selects the SAT solver.

Vampire handles theories by automatically adding theory axioms to the search space whenever an interpreted sort, function, or predicate is found in the input. This approach is incomplete for theories such as linear and non-linear integer and real arithmetic, but shows good results in practice. The `-tha` option of Vampire with values `on` and `off` controls whether theory axioms are added.

A recent work [66] lifted AVATAR to be modulo theories by replacing the SAT solver by an SMT solver, ensuring that the sub-problem is theory-consistent in the ground part. The result is that the saturation prover and the SMT solver deal with the parts of the problem to which they are best suited. Vampire implements AVATAR modulo theories using Z3.

Our experience with running Vampire on theory- and quantifier-intensive problems shows that some of the theory axioms can degrade the performance of Vampire. These axioms make Vampire infer many theory tautologies making the search space larger. We found that, among others, axioms of commutativity, associativity, left and right identity, and left and right inverse of arithmetic operations are in this sense "expensive". Our solution to this problem is a more refined control over which theory axioms Vampire adds to the search space. We added to the `-tha` option of Vampire a new value named `some` that makes Vampire only add "cheap" axioms to the search space. `some` implements our empirical criterion for choosing theory axioms. Designing other criteria for axiom selection is an interesting task for future work.

Table 4.1. Runtimes in seconds of Vampire on the Boogie translation of the benchmarks.

Benchmark	Number of loop unrollings				
	1	2	3	4	5
binary-search	0.884	2.420	3.364	10.709	27.648
bubble-sort	–	–	–	–	–
dutch-flag	24.789	–	–	–	–
insertion-sort	122.354	–	–	–	–
matrix-transpose	1.311	–	1.078	–	–
maxarray	0.205	0.587	1.197	1.702	1.692
maximum	0.066	0.078	0.082	0.095	0.129
one-duplicate	–	–	–	–	–
select-k	96.993	–	–	–	–
two-way-sort	0.191	0.205	0.647	1.384	1.344

4.4.4 Experimental Results

For our experiments, we compared the performance of Vampire on the Boogie, Voogie, and BLT translations of our benchmarks.

We ran Vampire on all three sets of benchmarks with options `-tha some` and `-sas z3`. Vampire supports both TPTP and SMT-LIB syntax, the input language is selected by setting the `--input_syntax` option to `tptp` and `smtlib2`, respectively. We performed our experiments on the StarExec compute cluster [78] using the time limit of 5 minutes per problem. The detailed experimental results are available at `http://www.cse.chalmers.se/~evgenyk/ijcar18/`.

Tables 4.1 and 4.2 summarise the results of Vampire on the Boogie and Voogie translations of the benchmarks, respectively. A dash means that Vampire does not solve the problem within the given time limit.

- Vampire solves 25 of the problems, translated by Boogie, and 36 problems, translated by Voogie.
- For 16 benchmark programs, Vampire solves their Voogie translations, but not the Boogie translations.
- For 5 benchmark programs, Vampire solves their Boogie translations, but not the Voogie translations.
- For 20 benchmark programs, Vampire solves both of their translations, and is faster on the Voogie translations for 12 of them.

Table 4.3 summarises the results of Vampire on the BLT translations of the benchmarks.

Table 4.2. Runtimes in seconds of Vampire on the Voogie translation of the benchmarks.

Benchmark	Number of loop unrollings				
	1	2	3	4	5
binary-search	1.979	25.135	6.560	–	163.803
bubble-sort	0.394	53.192	2.073	–	–
dutch-flag	11.384	–	–	–	–
insertion-sort	18.262	38.169	3.369	21.698	11.639
matrix-transpose	0.266	8.362	–	–	–
maxarray	0.170	0.587	0.489	2.635	6.325
maximum	0.062	0.065	0.070	0.087	0.102
one-duplicate	0.125	2.402	2.231	93.746	145.243
select-k	0.216	0.612	203.655	–	–
two-way-sort	0.464	5.360	–	–	–

- Vampire solves 19 of the problems, translated by BLT.
- For all benchmark programs whose BLT translation Vampire is able to solve, Vampire also solves their Voogie translations. There are 3 benchmark programs for which Vampire solves their BLT translations but not their Boogie translations.

Based on the results presented in Tables 4.1–4.3 we make the following observation. The problems translated from our benchmarks by Voogie are easier for Vampire than the problems translated by Boogie and BLT. Vampire is more efficient both in terms of the number of solved problems and runtime on the problems translated by Voogie. This confirms our conjecture that the use of (efficient translations of) FOOL is better for saturation theorem provers than translations to FOL designed for other purposes. It would be interesting to run these experiments for theorem provers other than Vampire, however Vampire is currently the only prover implementing FOOL.

4.5 Related Work

Our previous work introduced FOOL [50], its implementation in Vampire [48], and an efficient clausification algorithm for FOOL formulas [49].

In [48] we sketched a tuple extension of FOOL and an algorithm for computing the next state relations of imperative programs that uses this extension. This paper extends and improves the algorithm. In particular, (i) we described an encoding that uses FOOL in its current form, available

Table 4.3. Runtimes in seconds of Vampire on the BLT translation of the benchmarks.

Benchmark	Number of loop unrollings				
	1	2	3	4	5
binary-search	0.821	163.790	–	–	–
bubble-sort	3.511	–	–	–	–
dutch-flag	4.049	–	–	–	–
insertion-sort	1.780	–	–	–	–
matrix-transpose	0.465	12.437	–	–	–
maxarray	0.174	1.567	47.724	–	–
maximum	0.069	0.140	0.724	12.234	–
one-duplicate	0.307	10.039	–	–	–
select-k	3.142	–	–	–	–
two-way-sort	0.319	24.622	–	–	–

in Vampire, (ii) we refined the encoding to only use in `let-in` the variables updated in program statements, (iii) we gave the definition of the encoding formally and in full detail, and (iv) we presented experimental results that confirm the described benefits of the encoding.

Boogie is used as the name of both the intermediate verification language [57] and the automated verification framework [6]. The Boogie verifier encodes the next state relations of imperative programs in first-order logic by naming intermediate states of program variables [56].

BLT [23] is a tool that automatically generates verification conditions of Boogie programs. The aim of the BLT project is to use first-order theorem provers rather than SMT solvers for checking quantified program properties. BLT produces formulas written in the TPTP language and uses `if-then-else` and `let-in` constructs of FOOL. BLT has an experimental option that introduces tuples for encoding of the next state relation. This option implements the encoding described in our earlier work [48].

4.6 Conclusion and Future Work

We presented an encoding of the next state relations of imperative programs in FOOL. Based on this encoding we defined a translation from imperative programs, annotated with their pre- and post-conditions, to FOOL formulas that encode partial correctness properties of these programs. We presented experimental results obtained by running the theorem prover Vampire on such properties. We generated these properties

using our translation and verification tools Boogie and BLT. We described a polymorphic theory of first class tuples and its implementation in Vampire.

The formulas produced by our translation can be efficiently checked by automated theorem provers that support FOOL. The structure of our encoding closely resembles the structure of the program. The encoding contains neither new symbols nor duplicated parts of the program. This way, the efficient representation of the problem in plain first-order logic is left to the theorem prover that is better equipped to do it.

Our encoding is useful for automated program analysis and verification. Our experimental results show that Vampire was more efficient in terms of the number of solved problems and runtime on the problems obtained using our translation.

FOOL reduces the gap between programming languages and languages of automated theorem provers. Our encoding relies on tuple expressions and `let-in` with tuple definitions, available in FOOL. To our knowledge, these constructs are not available in any other logic efficiently implemented in automated theorem provers.

The polymorphic theory of first class tuples is a useful addition to a first-order theorem prover. On the one hand, it generalises and simplifies tuple expressions in FOOL. On the other hand, it is a convenient theory on its own, and can be used for expressing problems of program analysis and computer mathematics.

For future work we are interested in making automated first-order theorem provers friendlier to program analysis and verification. One direction of this work is design of an efficient translation of features of programming languages to languages of automated theorem provers. Another direction is extensions of first-order theorem provers with new theories, such as the theory of bit vectors. Finally, we are interested in further improving automated reasoning in combination of theories and quantifiers.

Acknowledgements

This work has been supported by the ERC Starting Grant 2014 SYMCAR 639270, the Wallenberg Academy Fellowship 2014, the Swedish VR grant D0497701, the Austrian research project FWF S11409-N23 and the EPSRC grant EP/P03408X/1-QuTie.

Chapter 5

Checking Network Reachability Properties by Automated Reasoning in First-Order Logic

Evgenii Kotelnikov and Pavle Subotić

Abstract. Verification of computer networks is an important and challenging task, often tackled with constraint solving and automated reasoning. This work presents an approach for checking and discovering reachability properties of virtual private cloud networks using automated reasoning in first-order logic. Reachability properties of a network express whether its configuration allows the network traffic to flow between given nodes of the network. We model networks with Horn clauses and check first-order properties of these models using Vampire both as a finite model builder and as a saturation theorem prover.

A technical report based on a joint work with Byron Cook, Temesghen Kahsai and Sean McLaughlin, 2018.

5.1 Introduction

Computer networks are typically built from a variety of specialised heterogeneous devices running complex distributed protocols. Network administrators, responsible for operability of a network, must configure and deploy every protocol separately on each individual device. In an effort to simplify this task, *software-defined networking* (SDN) [55] has been proposed as a modern alternative. SDN provides an layer of software that is installed on each of the network devices and a logically-centralised controller that manages connectivity settings of each device. Administrators can therefore manage networks at a higher level of abstraction by programming them from the controller, typically in a domain specific language [33]. The SDN architecture is often credited with flexibility and ease of maintenance compared to traditional networks [13].

Modern platforms for cloud computing such as Amazon Elastic Compute Cloud, Google Compute Engine and Azure Virtual Machines offer their users means of configuring *virtual private cloud* (VPC) networks in the style of SDN. Administrators of such networks use a centralised control panel or a specialised API to launch network nodes, set up subnets and route tables, and tune connectivity and security settings of the network. Despite the increase of usability provided by the cloud platforms, VPC networks remain prone to misconfigurations. These misconfigurations are caused by the complexity of large-scale enterprise networks and might lead to downtimes and breaches of security. Discovering such misconfigurations in industrial-sized networks is both labor-intensive and computationally hard.

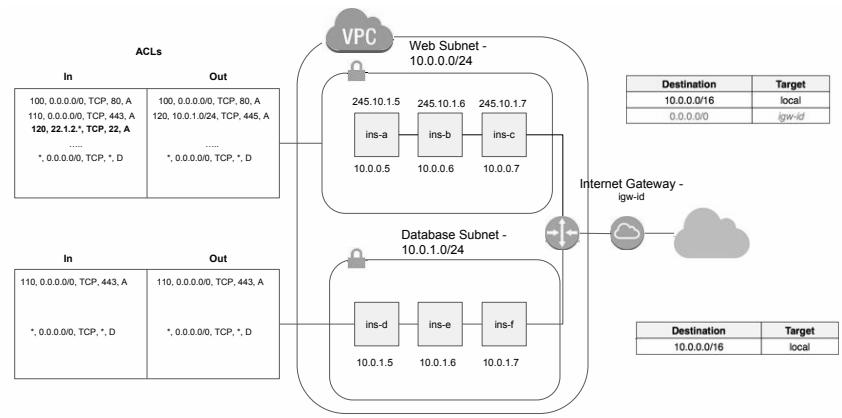
The presence of a centralised network configuration facilitates automated analysis and synthesis of SDN networks. Indeed, several tools have been developed [32, 42, 16, 5, 44, 1, 59, 30] in an effort to verify various SDN components. These tools employ specialised algorithms [44] as well as general purpose reasoning engines such as Datalog [39, 30, 32], BDD [1], SMT [42, 16] and SAT [59, 5].

Despite the vast body of available tools, the problem of practical verification of enterprise networks remains a challenge. Firstly, the size and intricacy of modern industrial-sized networks results in very difficult constraint satisfaction problems. While most network verification tools depend on a single constraint solving paradigm, we find that no single solver achieves best performance on all verification problems. Secondly, general purpose solvers are very performance-sensitive to problem encodings. Effective encodings are often a "black art" for users, resulting in

tedious trial and error for a user without a good understanding of the implementation of the underlying solver.

We are interested in checking properties of VPC networks using complementary constraint solvers. Our ultimate goal is to build a portfolio of constraint solvers that leverages the advantages of different types of solvers on different types of properties. In an effort towards this goal in this work we describe an approach to network verification that relies on automated reasoning for first-order logic. Our plan is to integrate this approach to our ongoing work on the portfolio of solvers.

We consider reachability properties that express whether the network traffic is able to flow between given nodes of a VPC network. The traffic flow in these networks is typically controlled by the rules assigned to various networking components provided by the cloud platform. These components include subnets, route tables, access control lists, internet gateways and others. Let us illustrate these properties using the following example of a VPC network.



This network consists of an internet gateway, two subnets "Web" and "Database" and three network nodes in each of them. Each of the subnets is assigned with a route table (on the right) and an access control list (ACL, on the left). The route tables allow the network traffic to flow between the subnets and between the "Web" subnet and the internet gateway. In other words, the network nodes in the "Web" subnet are accessible from the internet, while the nodes in the "Database" subnet are not. An ACL consists of rules that filter the network traffic to and from its subnet. In our example, one of the ACL rules of the "Database" subnet forbids SSH access to its nodes, both directly and through an intermediate node.

Imagine that this network grows over time and has more nodes and security and access rules added to it. A network administrator may want to make sure that the network retains certain properties after each change in its configuration. For example, the network administrator may want to check the following property.

Example 5.1. All network nodes in the subnet "Web" can access all network nodes in the subnet "Database".

The network administrator might also want to know which networking components satisfy a given property, such as the ones described in the following example.

Example 5.2. All network nodes that have the port 22 (SSH) accessible from the internet.

We will refer to questions that network administrators might want to answer, such as the ones in Examples 5.1 and 5.2, as *network questions*. In particular, we will refer to questions similar to Example 5.1 as *boolean questions*, because an answer to them is "yes" or "no", and to questions similar to Example 5.2 as *list questions*, because an answer to them is a list of networking components. Each boolean question can be equivalently phrased as a list question such that the answer to the boolean question is "yes" iff the answer to the correspondent list questions is not empty. However, we distinguish these two types of questions because, as we show later, we can more efficiently answer them using different techniques.

Answering network questions manually might be tedious and error-prone in an industrial-size network. For this reason it is necessary to automate this task with specialised tools. In this work we employ automated theorem provers for first-order logic. To this end, we build static models of VPC networks (Section 5.2), translate these models and network questions about them to problems in first-order logic (Section 5.4) and check these problems using finite model builders and saturation-based theorem provers (Section 5.3). In Section 5.3 we argue that finite model builders can more efficiently answer list questions and saturation-based theorem prover can more efficiently answer boolean questions. In Section 5.5 we cite related work and in Section 5.6 we describe the challenges that we faced in this work and outline future work.

5.2 Network Reachability Properties

We answer network questions statically, that is, instead of sending packets in a network, we build a static model of the network and reason about

properties of this model. Our *network model* consists of two parts, the *formal specification* and the *snapshot* of the network. The specification formalises the semantics of each of the components available in the network. For example, the formal specification describes how a route table directs network traffic in a subnet or in which order a firewall applies rules in an access control list. The snapshot describes the topology of the given network. For example, the snapshot contains the list of network nodes, subnets and their route tables. Naturally, the formal specification in the model of each particular VPC network is the same, whereas the snapshot differs. We used models of Amazon VPC networks as part of our ongoing work on network verification using complementary constraint solvers. We express network questions in the language of many-sorted first-order logic. In this section we describe syntax and semantics of network models and network questions.

5.2.1 Network Models

A network model is a finite set of first-order Horn clauses expressed in a logic programming style. We disallow function symbols with positive arity and allow stratified negation. We assume the plain logic programming semantics for these Horn clauses, defined in the standard way (see e.g. [58]). In particular, we make the closed-world assumption and treat negation as failure. In addition, our network models use the theory of bit vectors to describe ports, port ranges, IPv4 addresses and subnet masks.

A *signature* of the network model is a triple (T, C, P), where T is a set of *types*, C is a set of *constants* and P is a set of *predicates*. We assign each constant with a type $\tau \in T$ and each predicate with a type $\tau_1 \times \ldots \times \tau_n$ $(n \geq 0)$, where $\tau_i \in T$ for each $1 \leq i \leq n$. We assume a countable infinite set of *variables*. We assign each variable with a type $\tau \in T$. We call a *term* of the type $\tau \in T$ a constant or a variable of that type. We call an *atom* an expression of the form $p(t_1, \ldots, t_n)$, where $n > 0$, $p \in P$ is a predicate of the type $\tau_1 \times \ldots \times \tau_n$ and each t_i, $1 \leq i \leq n$ is a term of the type τ_i. We call a *literal* an atom or its negation.

A *rule* is a Horn clause of the form $A \leftarrow L_1 \wedge \ldots \wedge L_n$ $(n \geq 0)$, where the *head* of the rule A is an atom and each of $L_1, \ldots, L_n$ is a literal. If $n = 0$ and all arguments of A are constants then we call such rule a fact. We call a *definition* of the predicate $p \in P$ the set of all rules in the network model that use p in their head.

The specification part of the model contains types, constants, predicates and rules that describe the semantics of the networking components used in the network. For example, the specification defines the semantics

of SSH tunneling. One network node can SSH tunnel to another node iff it can either connect to it over SSH directly, or through a chain of one or more intermediate nodes. In order to express this concept, the specification contains predicates *canSshTunnel* and *canSsh*, each of the type node × node, and the two following rules.

$$\begin{aligned} \mathit{canSshTunnel}(\mathit{Node}_1, \mathit{Node}_2) &\leftarrow \mathit{canSsh}(\mathit{Node}_1, \mathit{Node}_2). \\ \mathit{canSshTunnel}(\mathit{Node}_1, \mathit{Node}_2) &\leftarrow \mathit{canSshTunnel}(\mathit{Node}_1, \mathit{Node}_3) \\ &\quad \wedge\ \mathit{canSshTunnel}(\mathit{Node}_3, \mathit{Node}_2). \end{aligned}$$

The snapshot part of the model contains constants and facts that describe the configuration of the networking components in a given network. For example, the snapshot of a network with a single node `i-abcd1234` in a single subnet "Web" consists of the constants $\mathrm{node}_{\mathrm{abcd1234}}$ and $\mathrm{subnet}_{\mathrm{Web}}$, and the fact $\mathit{nodeHasSubnet}(\mathrm{node}_{\mathrm{abcd1234}}, \mathrm{subnet}_{\mathrm{Web}})$.

We assume that the signature contains (i) types bits16 and bits32; (ii) 2^{16} constants of the type bits16; (iii) 2^{32} constants of the type bits32; (iv) predicates $\mathit{bits16}_{\leq}$ and $\mathit{bits16}_{\geq}$ of the type bits16 × bits16 with a special semantics; and (v) predicate $\mathit{bits32}_{\wedge}$ or the type bits32×bits32×bits32 with a special semantics. bits16 and bits32 represent the types of 16-bit and 32-bit vectors. The semantics of the predicates is that of the correspondent operations over bit vectors defined in the standard way.

The network specification uses 16-bit vectors to encode port numbers (as integers between 0 and 65535) and port ranges, and 32-bit vectors to encode IPv4 addresses and subnet masks. Port ranges are represented using the type portRange and the predicate *portRange* of the type portRange × bits16 × bits16. For example, the port range 0–1023 is represented as the constant $\mathrm{portRange}_{0\text{–}1023}$ of the type portRange and the fact $\mathit{portRange}(\mathrm{portRange}_{0\text{–}1023}, \mathrm{bits16}_0, \mathrm{bits16}_{1023})$, where $\mathrm{bits16}_0$ and $\mathrm{bits16}_{1023}$ are constants of the type bits16. The network specification contains the following definition of the predicate *portRangeOverlap* of the type portRange × portRange that holds when two port ranges overlap.

$$\begin{aligned} \mathit{portRangeOverlap}(\mathit{Range}_1, \mathit{Range}_2) \leftarrow\ & \mathit{portRange}(\mathit{Range}_1, \mathit{From}_1, \mathit{To}_1) \\ \wedge\ & \mathit{portRange}(\mathit{Range}_2, \mathit{From}_2, \mathit{To}_2) \\ \wedge\ & \mathit{bits16}_{\leq}(\mathit{From}_1, \mathit{To}_2) \\ \wedge\ & \mathit{bits16}_{\leq}(\mathit{From}_2, \mathit{To}_1). \end{aligned}$$

Finally, we assume that for each type $\tau \in T$ the signature contains the equality predicate $=_\tau$ of the type $\tau \times \tau$ and the network model contains the rule $=_\tau(X, X)$.

The specification of Amazon VPC networks that we used in this work consists of approximately 50 types, 200 predicates and over 240 rules.

5.2.2 Network Questions

We express network questions as formulas of many-sorted first-order logic with the standard logical connectives $\vee$, $\wedge$, $\Rightarrow$, $\Leftrightarrow$, $\oplus$ and equality. These formulas only use types, constants and predicates from the signature of the network model. The formulas do not use any function symbols. We allow interpretation of these formulas to use empty domains and otherwise assume the standard semantics of many-sorted first-order logic.

We express boolean questions as closed formulas, that is, formulas in which all variables are bound by a quantifier. Conversely, we express list questions as formulas with free variables. The answer to a boolean question is "yes" iff its correspondent formula is valid. The answer to a list question is the set of substitutions of free variables with constants that satisfy its correspondent formula.

Formula 5.1 expresses the boolean question in Example 5.1.

$$\begin{aligned}
&(\forall w : \text{node})(\forall d : \text{node}) \\
&\quad (\mathit{nodeHasSubnet}(w, \text{subnet}_{\text{Web}}) \wedge \\
&\quad\ \mathit{nodeHasSubnet}(d, \text{subnet}_{\text{Database}}) \Rightarrow \\
&\quad\quad \mathit{nodeCanConnectToNode}(w, d))
\end{aligned} \tag{5.1}$$

Formula 5.2 expresses the list question in Example 5.2. In this formula n of the type node and e of the type eni are free variables.

$$\begin{aligned}
&\mathit{nodeHasEni}(n, e) \wedge \\
&\mathit{reachablePublicTcpUdp}(\text{dir}_{\text{ingress}}, \text{proto}_6, e, \text{port}_{22}, \\
&\qquad\qquad\qquad\qquad \text{publicIp}_{8:8:8:8}, \text{port}_{40000})
\end{aligned} \tag{5.2}$$

All predicates and constants used in Formulas 5.1 and 5.2 are part of the signature of the network model. Constants $\text{subnet}_{\text{Web}}$ and $\text{subnet}_{\text{Database}}$ are part of the network snapshot, and all other predicates and constants are part of the network specification.

5.3 Checking Properties with Theorem Provers

We translate network models and network questions to problems in first-order logic, some of which use theories. We solve these problems using saturation-based theorem provers and finite model builders. We use the combination of these two types of provers to leverage the strengths of both of them, which we summarize below.

Saturation-based theorem provers such as E [75], Spass [95] or Vampire [54] construct proofs of unsatisfiability of first-order problems. To that end, they first convert the input problem into a set of first-order clauses and then try to derive contradiction from it. Theorem provers saturate the search space by inferring new clauses with inference rules such as binary resolution [4] and superposition [63]. They employ multiple techniques to prune the search space such as simplification orderings, selection functions and redundancy elimination. Saturation-based provers handle theories by adding incomplete first-order theory axioms to the set of clauses and by using specialised inference rules. In addition to that, Vampire implements the AVATAR modulo theories [66] architecture that relies on an SMT solver for theory-consistent reasoning in the ground subset of the problem. Saturation-based provers are designed to efficiently solve unsatisfiable problems. Given a satisfiable problem, saturation-based provers can in rare cases report satisfiability and output the saturated set of clauses. However, it is usually not possible to reconstruct a model from this set.

Finite model builders (finders) construct finite counter-models of first-order problems. One of the mots successful methods for finite model building was pioneered by the finite model builder MACE [60]. This method iterates over possible domain sizes, for each domain size grounds the first-order problem with this domain, and translates the resulting formula to a SAT problem. If the SAT problem is satisfied, a finite model of the selected size is reconstructed from the SAT model. Finite model builders Gandalf [89], Paradox [24] and Vampire [69] implement the MACE-style method. Finite model builders generally do not support theories with the notable exception of SMT solver CVC4 [8] that integrates model finding techniques into theory reasoning [72]. Finite model builders are designed to efficiently solve satisfiable problems. Given an unsatisfiable problem, some finite model builders might detect that the problem cannot have infinite models and in such case report unsatisfiability.

We translate (i) each boolean question to a first-order problem that is unsatisfiable iff the answer to the question is true; and (ii) each list question to a first-order problem such that each of its finite models cor-

Output of a theorem prover	Boolean question	List question
Saturation found unsat	yes	empty list
Saturation found sat	no	error
FMB found unsat	yes	empty list
FMB found a model	no	list of answers

Table 5.1. Solutions found by saturation-based theorem provers and finite model builders (FMB) interpreted as answers to network questions.

responds to an answer to the network question. Table 5.1 summarises how we interpret a solution found by a theorem prover as the answer to the network question. Saturation-based theorem provers generally cannot answer list questions except for the degenerate case when the answer is empty. With this exception, both types of provers are able to answer both types of network questions.

We run a finite model builder and a saturation-based prover in parallel on each problem and record the result of the fastest successful run. We expect that in most cases (i) a boolean question is answered with "yes" by the saturation-based prover and with "no" by the finite model builder; and (ii) a list question is answered with the empty list by the saturation-based prover and with a non-empty list by the finite model builder.

In this work we used the Vampire theorem prover both as a saturation-based theorem prover and a finite model builder. Our translation produces problems expressed in a logic supported by Vampire, namely many-sorted first-order logic with equality, extended with the theory of linear integer arithmetic, the theory of arrays [48] and the theory of tuples [51]. We wrote the problems in the TPTP language [79].

Our problems use the first-order formula φ that expresses the network question and first-order axioms $A_1, \ldots, A_n$ that we translate from the network model. We translate each boolean question to a problem of the form $A_1 \wedge \ldots \wedge A_n \Rightarrow \neg\varphi$ and each list question to a problem of the form $A_1 \wedge \ldots \wedge A_n \wedge (\forall \bar{z})(q(\bar{z}) \Leftrightarrow \varphi) \Rightarrow (\forall \bar{z})q(\bar{z})$, where q is a fresh predicate symbol and $\bar{z}$ are fresh free variables of φ. We reconstruct the answer to a list question from the model of the q predicate — each substitution of $\bar{z}$ that satisfies q is an answer to the question.

5.4 Network Reachability as First-Order Problem

Our translation of network models to problems in first-order logic consists of translations of types, constants, predicate definitions and theories. We translate types, constants (Section 5.4.1) and predicate definitions (Sec-

tion 5.4.2) using Clark completion [25]. We use specialised translations for the theory of bit vectors (Section 5.4.3) because Vampire does not support it.

5.4.1 Types and Constants

Let τ be a type and $c_1, \ldots, c_n$ $(n \geq 0)$ be constants of this type. If $n > 0$ then we introduce a sort τ, constants $c_1, \ldots, c_n$ of this sort and add the domain closure axiom of the form $(\forall x : \tau)(x = c_1 \vee \ldots \vee x = c_n)$ and the distinct constants axiom of the form of a conjunction of literals $c_i \neq c_j$ for all $1 \leq i \leq n$, $1 \leq j \leq n$ and $i \neq j$. If $n = 0$ then we do not introduce any sorts or constants and in our translation of predicate definitions replace each subformula of the form $(\forall x : \tau)\varphi$ with logical truth and each subformula of the form $(\exists x : \tau)\varphi$ with logical falsum.

For example, the signature contains the type dir and two constants $\mathrm{dir}_{\mathrm{ingress}}$ and $\mathrm{dir}_{\mathrm{egress}}$ of this type. dir represents the direction of a network package. We translate this type to first-order logic as a sort *dir*, two constants *ingress* and *egress* of this sort and axioms $(\forall x : \mathit{dir})(x = \mathit{ingress} \vee x = \mathit{egress})$ and $\mathit{ingress} \neq \mathit{egress}$.

5.4.2 Predicate Definitions

Let predicate p of the type $\tau_1 \times \ldots \times \tau_n$ be defined using $k \geq 0$ rules. Let $x_1 : \tau_1, \ldots, x_n : \tau_n$ be fresh variables. If $k = 0$ then we translate the definition of p to the axiom

$$(\forall x_1 : \tau_1) \ldots (\forall x_n : \tau_n)(\neg p(x_1, \ldots, x_n)).$$

If $k > 0$ then we translate the definition of p to the axiom

$$(\forall x_1 : \tau_1) \ldots (\forall x_n : \tau_n)(p(x_1, \ldots, x_n) \Leftrightarrow R_1 \vee \ldots \vee R_k),$$

where each of the formulas $R_1, \ldots, R_k$ are translations of each of the k rules, respectively.

Let a rule be of the form

$$p(t_1, \ldots, t_n) \leftarrow L_1 \wedge \ldots \wedge L_m, \quad (m \geq 0)$$

where $t_1 : \tau_1, \ldots, t_n : \tau_n$ are terms. Let $y_1 : \sigma_1, \ldots, y_l : \sigma_l$ be variables that occur in either of $L_1, \ldots, L_m$ but not in $p(t_1, \ldots, t_n)$. If $m = 0$ then we translate the rule to the formula $x_1 = t_1 \wedge \ldots \wedge x_n = t_n$. If $m > 0$ then

we translate the rule to the formula

$$x_1 = t_1 \wedge \ldots \wedge x_n = t_n \ \wedge (\exists y_1 : \sigma_1) \ldots (\exists y_l : \sigma_l)(L_1 \wedge \ldots \wedge L_m).$$

For example, consider the following definition of the predicate *link* of the type node $\times$ node that encodes the equivalence relation between two nodes. The definition of *link* in not present in the network model, but it is illustrative of our translation.

$$\begin{aligned} &\mathit{link}(X, X). \\ &\mathit{link}(X, Y) \leftarrow \mathit{link}(Y, X). \\ &\mathit{link}(X, Y) \leftarrow \mathit{link}(X, Z) \wedge \mathit{link}(Z, Y). \end{aligned}$$

We translate the definition of *link* as a predicate symbol $\mathit{Link} : \mathit{node} \times \mathit{node}$ and the axiom

$$\begin{aligned} &(\forall x : \mathit{node})(\forall y : \mathit{node}) \\ &\quad (\mathit{Link}(x, y) \Leftrightarrow x = y \vee \\ &\qquad\qquad \mathit{Link}(y, x) \vee \\ &\qquad\qquad (\exists z : \mathit{node})(\mathit{Link}(x, z) \wedge \mathit{Link}(z, y))). \end{aligned}$$

Our translation ignores definitions of equality predicates and instead uses the standard equality in first-order logic.

5.4.3 Theories

Our early experiments with translating the theory of bit vectors revealed that the quality of this translation crucially affects the performance of Vampire. This led us to develop and compare multiple translations. In this section we first describe translations of bit vectors that use other theories supported by Vampire. Then, we describe a model transformation that eliminates theories from network models altogether by precomputing the results of theory operations. Finally, we compare these translations.

Translations to Other Theories

Recall that our network models use (i) the type bits16 and predicates $\mathit{bits16}_{\leq}$ ("less or equal"), $\mathit{bits16}_{\geq}$ ("greater or equal") and $=_{\text{bits16}}$, and (ii) the type bits32 and predicates $\mathit{bits32}_{\wedge}$ (bitwise conjunction) and $=_{\text{bits32}}$. We translate 16-bit vectors to integers and 32-bit vectors to tuples and arrays.

16-bit vectors to integers. We translate (i) the type bits16 to $\mathbb{Z}$, (ii) each of the 16-bit vector constants to an integer constant, and (iii) each of the 16-bit predicates to their correspondent predicate symbol in the theory of linear integer arithmetic. (iv) predicate $=_{\text{bits16}}$ to the standard equality.

32-bit vectors to tuples. We translate (i) the type bits32 to the tuple sort $(bool, \ldots, bool)$ where *bool* is repeated 32 times, (ii) each of the 32-bit vector constants to a term $(b_1, \ldots, b_{32})$ of this sort, where each of $b_1, \ldots, b_{32}$ is either *true* or *false*, and (iii) predicate $bits32_\wedge$ to a predicate defined by the following axiom of bitwise conjunction for boolean tuples.

$$\begin{aligned}&(\forall x_1 \ldots x_{32} : bool)(\forall y_1 \ldots y_{32} : bool)(\forall z_1 \ldots z_{32} : bool)\\ &\quad (bits32_\wedge((x_1, \ldots, x_{32}), (y_1, \ldots, y_{32}), (z_1, \ldots, z_{32})) \Leftrightarrow\\ &\qquad (z_1 \Leftrightarrow x_1 \wedge y_1) \wedge \ldots \wedge (z_{32} \Leftrightarrow x_{32} \wedge y_{32}))\end{aligned}$$

(iv) predicate $=_{\text{bits32}}$ to the standard equality.

32-bit vectors to arrays. We translate (i) the type bits32 to the array sort $array(\mathbb{Z}, bool)$, (ii) each of the 32-bit vector constant to a fresh constant v of this sort, defined using the axiom $select(v, 1) = b_1 \wedge \ldots \wedge select(v, 32) = b_{32}$, where each of $b_1, \ldots, b_{32}$ is either *true* or *false*, and (iii) predicate $bits32_\wedge$ to a predicate defined by the axiom of bitwise conjunction for arrays of booleans.

$$\begin{aligned}&(\forall x : array(\mathbb{Z}, bool))(\forall y : array(\mathbb{Z}, bool))(\forall z : array(\mathbb{Z}, bool))\\ &\quad (bits32_\wedge(x, y, z) \Leftrightarrow (select(z, 1) \Leftrightarrow select(x, 1) \wedge select(y, 1)) \wedge\\ &\qquad\qquad \ldots\\ &\qquad\qquad \wedge (select(z, 32) \Leftrightarrow select(x, 32) \wedge select(y, 32)))\end{aligned}$$

(iv) predicate $=_{\text{bits32}}$ to a predicate defined by the following axiom of equality for 32-element arrays of booleans.

$$\begin{aligned}&(\forall x : array(\mathbb{Z}, bool))(\forall y : array(\mathbb{Z}, bool))\\ &\quad (=_{\text{bits32}}(x, y) \Leftrightarrow select(x, 1) = select(y, 1) \wedge\\ &\qquad\qquad \ldots\\ &\qquad\qquad \wedge select(x, 32) = select(y, 32))\end{aligned}$$

Precomputation of Theories

Our network models use 16-bit vectors to describe port numbers and port ranges and 32-bit vectors to describe IPv4 addresses and subnet masks. Bit vector predicates are only used in the definitions of predicates for network primitives. For example, 16-bit vectors only occur in the definitions of predicates that encode set-theoretic operations over port ranges such as *portRangeOverlap* mentioned in Section 5.2.2. Definitions of higher level predicates only use these operations over port ranges and never the bit vectors predicates.

We eliminate theories from network models by rewriting the definitions of set-theoretic operations over network primitives with results of precomputation of these operations for all primitives used in the network. For example, we precompute the *portRangeOverlap* predicate in the following way. First, we compute the set S of all pairs of overlapping port ranges among the ones used in the network. This requires a quadratic number of integer comparisons. Then, we use set S to build a definition of *portRangeOverlap* that consists of facts of the form $\mathit{portRangeOverlap}(\text{portRange}_X, \text{portRange}_Y)$ where $(X, Y) \in S$. We optimise precomputation of some operations by considering their algebraic properties. For example, *portRangeOverlap* is a reflexive symmetric binary relation and 0–65535 overlaps with any other port range. Therefore, we avoid comparing each port range with itself, comparing two port ranges twice and comparing 0–65535 with other port ranges. Instead, we use the following rules in the definition of *portRangeOverlap*.

$$\begin{aligned} &\mathit{portRangeOverlap}(\mathit{Range}, \mathit{Range}). \\ &\mathit{portRangeOverlap}(R_1, R_2) \leftarrow \mathit{portRangeOverlap}(R_2, R_1). \\ &\mathit{portRangeOverlap}(\text{portRange}_{0\text{–}65535}, \mathit{Range}). \end{aligned} \tag{5.3}$$

Consider a network that only uses port ranges 0–65535, 22–22, 443–443, 5432–5432, 80–80 and 80–81 in any of the configurations of its components. We found that it is not uncommon for our networks to use trivial port ranges that span one port number. The definition of *portRangeOverlap* precomputed for this network consists of 12 facts or the rule $\mathit{portRangeOverlap}(\text{portRange}_{80\text{–}80}, \text{portRange}_{80\text{–}81})$ together with the rules (5.3).

Comparison of Translations

We observed that formulas produced by the translation of 32-bit vectors to tuples are generally easier for Vampire than formulas produced by the

translation to arrays. We explain it by the fact that the former end up using a more compact representation of bit vectors in first-order logic. The translation to tuples models a 32-bit vector as a single term, while the translation to arrays models it as conjunction of 32 literals. Furthermore, the translation to tuples uses the standard equality that Vampire supports efficiently, while the translation to arrays uses a heavy equality axiom for 32-element arrays. The compact representation of bit vectors as tuples allows Vampire infer bitwise conjunctions faster.

We observed that formulas produced by the translation with precomputed theories are generally easier for Vampire than formulas produced by any of the translations of bit vectors to other theories. We explain it by the fact that the bulky axioms for bitwise conjunction and equality slow down the proof search, whereas these operations are easy to precompute. Although the precomputation of a theory operation has polynomial complexity, we found that for the networks we used in this study it does not significantly slow down answering network questions. Realistic VPC networks only use a small number of distinct network primitives such as port numbers and IP addresses in any of the configurations of its components and the precomputation for them is not heavy. Furthermore, the precomputation of each theory operation usually involves a simple operation over machine integers that can be performed quickly.

Overall we found the precomputation of theories to be the most efficient strategy for translating our network models. It is also the only applicable strategy for translating network models for Vampire's finite model builder, because it only supports plain many-sorted first-order logic without theories.

5.5 Related Work

Network verification is an attractive target for automated reasoning tools, some of which we mentioned earlier in Section 5.1. In this section we focus on approaches to network verification that are most similar to this work and applications of finite model building and saturation theorem proving to similar problems.

Bjørner et al. [16] checked properties of ACLs, routing tables and Border Gateway Protocol policies in Microsoft Azure networks using SMT solver Z3 [27]. To that end [16] implemented a translation of network properties to quantifier-free logical formulas over bit vectors that is aware of the semantics of network components. In comparison, our translation encodes the complete semantics of a VPC network in the quantified

problem solved by the theorem prover. We argue that our approach is more scalable at the expense of having to deal with more difficult problems.

Weidenbach [94] and Goubault-Larrecq [34] used finite model building for first-order logic to check unrechability in security protocols.

5.6 Challenges and Future Work

This paper presents a work in progress and some of the challenges that we faced are yet to be overcome. We discuss these challenges in this section and suggest directions for future work. We believe that similar challenges are experienced in other practical applications of automated reasoning in first-order logic.

Good translation to first-order logic. Automated theorem provers for first-order logic are known to be sensitive to the encoding of the problem they are solving. To ensure efficient representation of practical problems in a theorem prover, these problems should be encoded using theories and syntactical constructs beyond plain first-order logic. Yet not all automated theorem provers efficiently implement these features. While the Vampire theorem prover used in this work supports many useful theories, we miss a support for the theory of bit vectors. The need to work around this limitation hinders a good translation of our network model to first-order logic.

Configuration of theorem provers. Automated theorem provers typically offer many settings for configuring proof search. For example, Vampire has dozens of options [68] that describe a colossal number of possible proof strategies. There is no proof strategy that would be the best for all kinds of problems, and we found that changing the default options of Vampire might result in a strategy that works better on our problems than the default strategy. At the same time, hand-picking good options is tedious and unintuitive, and there is no way of knowing if a hand-picked strategy will work well on all of our problems. Many modern automated theorem provers rely not on one single strategy, but on a portfolio of strategies, which they try one by one during proof search. Theorem provers typically implement portfolios for existing large collections of problems such as TPTP [79] and SMT-LIB [10]. To our knowledge, there is no systematic way of assembling a custom portfolio tailored to a specific class of problems.

Finite domains. Our formalisation of VPC networks uses large finite domains to describe the topology of a network and the semantics of its components. While finite domains is good news for finite model builders, saturation-based theorem provers are known to have hard time reasoning with them. Domain closure axioms $(\forall x : \tau)(x = c_1 \vee \ldots \vee x = c_n)$ for large values of n result in long clauses that degrade performance of the prover. A technique for efficient superposition theorem proving with these axioms is discussed e.g. in [38], but this technique is not implemented in any modern theorem prover.

EPR. We translate network models to problems that almost fit into the effectively propositional (EPR) fragment of first-order logic, also called the Bernays-Schönfinkel class. These problems do not use function symbols with positive arity, but skolem functions with positive arity might be introduced during clausification. The satisfiability problem for EPR is decidable and there exist efficient tools that deal with this fragment [47]. It is possible to translate our network models to problems in EPR, for example by grounding the existentially-quantified variables. The obvious drawback of this translation is the blowup of the size of the resulting problem. Whether this blowup is mitigated by the efficiency of EPR solvers is an interesting question for future work.

Solver-agnostic network models. The work presented in this paper contributes to our ultimate goal of checking network properties with diverse complementary constraint solvers. In order to succesfully use different kinds of constraint solvers we need a formalisation of networks that can be equally efficiently translated to different kinds of constraint satisfaction problems. We find that our current formalisation, written in the logic programming style, is friendly for systems like Datalog [35], but not necessarily for theorem provers for first-order logic. For example, some of the predicates in the specification can be more naturally expressed as functions in full first-order logic rather than translated from their encoding as Horn clauses. Ultimately this bias hinders efficient representation of the problem in first-order logic. Designing a more solver-agnostic formalisation of networks is an important direction of future work.

Evaluation. In this work we used a small collection of network configurations that we verified using Vampire playing a role of both a saturation-based theorem prover and a finite model builder. We leave for future work experiments with (i) other theorem provers for first-order logic, (ii) other

kinds of constraint solvers, (iii) different encodings of network problems as constraint satisfaction problems, and (iv) large collections of real world network configurations.

Acknowledgments

This work has been partially carried out during the first author's visit to Amazon Web Services. We thank Koen Claessen for his valuable comments on an earlier draft of this paper. The first author was partially supported by the Wallenberg Academy Fellowship 2014 and the Swedish VR grant D0497701.

Chapter 6

TFX: The TPTP Extended Typed First-Order Form

Geoff Sutcliffe and Evgenii Kotelnikov

Abstract. The TPTP world is a well established infrastructure that supports research, development, and deployment of Automated Theorem Proving systems for classical logics. The TPTP language is one of the keys to the success of the TPTP world. Originally the TPTP world supported only first-order clause normal form (CNF). Over the years support for full first-order form (FOF), monomorphic typed first-order form (TF0), rank-1 polymorphic typed first-order form (TF1), monomorphic typed higher-order form (TH0), and rank-1 polymorphic typed higher-order form (TH1), have been added. TF0 and TF1 together form the TFF language family; TH0 and TH1 together form the THF language family. This paper introduces the eXtended Typed First-order form (TFX), which extends TFF to include boolean terms, tuples, conditional expressions, and let expressions.

Published in the *Proceedings of the 6th Workshop on Practical Aspects of Automated Reasoning*. Springer, 2018.

6.1 Introduction

The TPTP world [81] is a well established infrastructure that supports research, development, and deployment of Automated Theorem Proving (ATP) systems for classical logics. The TPTP world includes the TPTP problem library, the TSTP solution library, standards for writing ATP problems and reporting ATP solutions, tools and services for processing ATP problems and solutions, and it supports the CADE ATP System Competition (CASC). Various parts of the TPTP world have been deployed in a range of applications, in both academia and industry. The web page `http://www.tptp.org` provides access to all components.

The TPTP language is one of the keys to the success of the TPTP world. The language is used for writing both TPTP problems and TSTP solutions, which enables convenient communication between different systems and researchers. Originally the TPTP world supported only first-order clause normal form (CNF) [87]. Over the years support for full first-order form (FOF) [80], monomorphic typed first-order form (TF0) [86], rank-1 polymorphic typed first-order form (TF1) [18], monomorphic typed higher-order form (TH0) [84], and rank-1 polymorphic typed higher-order form (TH1) [43], have been added. TF0 and TF1 together form the TFF language family; TH0 and TH1 together form the THF language family. See [83] for a recent review of the TPTP.

Since the inception of TFF there have been some features that have received little use, and hence little attention. In particular, tuples, conditional expressions (`if-then-else`), and let expressions (`let-in`) were neglected, and the latter two were horribly formulated with variants to distinguish between their use as formulae and terms. Recently, conditional expressions and let expressions have become more important because of their use in software verification applications. In an independent development, Evgenii Kotelnikov et al. introduced FOOL [50], a variant of many-sorted first-order logic (FOL). FOOL extends FOL in that it (i) contains an interpreted boolean type, which allows boolean variables to be used as formulae, and allows all formulae to be used as boolean terms, (ii) contains conditional expressions, and (iii) contains let expressions. FOOL can be straightforwardly extended with the polymorphic theory of tuples that defines first class tuple types and terms [51]. Features of FOOL can be used to concisely express problems coming from program analysis [51] or translated from more expressive logics. The conditional expressions and let expressions of FOOL resemble those of the SMT-LIB language version 2 [10].

The TPTP's new eXtended Typed First-order form (TFX) language remedies the old weaknesses of TFF, and incorporates the features of FOOL. This has been achieved by conflating (with some exceptions) formulae and terms, removing tuples from plain TFF, including fully expressive tuples in TFX, removing the old conditional expressions and let expressions from TFF, and including new elegant forms of conditional expressions and let expressions as part of TFX. (These more elegant forms have been mirrored in THF, but that is not a topic of this paper.) TFX is a superset of the revised TFF language. This paper describes the extensions to the TFF language form that define the TFX language. The remainder of this paper is organised as follows: Section 6.2 reviews the TFF language, and describes FOOL. Section 6.3 provides technical and syntax details of the new features of TFX. Section 6.4 describes the evolving software support for TFX, and provides some examples that illustrate its use. Section 6.5 concludes.

6.2 The TFF Language and FOOL

The TPTP language is a human-readable, easily machine-parsable, flexible and extensible language, suitable for writing both ATP problems and solutions. The top level building blocks of the TPTP language are *annotated formulae*. An annotated formula has the form

language`(`*name*`,` *role*`,` *formula*`,` `[`*source*`,` `[`*useful_info*`]]).`

The *language*s supported are clause normal form (`cnf`), first-order form (`fof`), typed first-order form (`tff`), and typed higher-order form (`thf`). The *role*, e.g., `axiom`, `lemma`, `conjecture`, defines the use of the formula in an ATP system. In the *formula*, terms and atoms follow Prolog conventions, i.e., functions and predicates start with a lowercase letter or are 'single quoted', variables start with an uppercase letter, and all contain only alphanumeric characters and underscore. The TPTP language also supports interpreted symbols, which either start with a `$`, or are composed of non-alphanumeric characters, e.g., the truth constants `$true` and `$false`, and integer/rational/real numbers such as `27`, `43/92`, `-99.66`. The basic logical connectives are `!`, `?`, `~`, `|`, `&`, `=>`, `<=`, `<=>`, and `<~>`, for $\forall$, $\exists$, $\neg$, $\vee$, $\wedge$, $\Rightarrow$, $\Leftarrow$, $\Leftrightarrow$, and $\oplus$ respectively. Equality and inequality are expressed as the infix operators `=` and `!=`. The following is an example of an annotated first-order formula, supplied from a file.

```
fof(union, axiom,
    ![X, A, B]: (member(X, union(A, B)) <=>
      (member(X, A) | member(X, B))),
    file('SET006+0.ax', union),
    [description('Definition of union'), relevance(0.9)]).
```

6.2.1 The Typed First-Order Form TFF

TFF extends the basic FOF language with *types* and *type declarations*. The TF0 variant is monomorphic, and the TF1 variant is rank-1 polymorphic. Every function and predicate symbol is declared before its use, with a *type signature* that specifies the types of the symbol's arguments and result. Each TF0 type is one of

- the predefined types `$i` for ι (individuals) and `$o` for o (booleans);
- the predefined arithmetic types `$int` (integers), `$rat` (rationals), and `$real` (reals); or
- user-defined types (constants).

User-defined types are declared before their use to be of the kind `$tType`, in annotated formulae with the `type` role — see Figure 6.1 for examples. Each TF0 type signature declares either

- an individual type τ; or
- a function type $(\tau_1 * \cdots * \tau_n) > \tilde{\tau}$ for $n > 0$, where τ_i are the argument types, and $\tilde{\tau}$ is the result type.

The type signatures of uninterpreted symbols are declared like types, in annotated formulae with the `type` role — see Figure 6.1 for examples. The type of `=` and `!=` is ad hoc polymorphic over all types except `$o` (this restriction is lifted in TFX), with both arguments having the same type and the result type being `$o`. The types of arithmetic predicates and functions are ad hoc polymorphic over the arithmetic types; see [86] for details. Figure 6.1 illustrates some TF0 formulae, whose conjecture can be proved from the axioms (it is the TPTP problem `PUZ130_1.p`).

The polymorphic TF1 extends TF0 with (user-defined) *type constructors*, *type variables*, polymorphic symbols, and one new binder. Each TF1 type is one of

- the predefined types `$i` and `$o`;

```
tff(animal_type, type, animal: $tType).
tff(cat_type, type, cat: $tType).
tff(dog_type, type, dog: $tType).
tff(human_type, type, human: $tType).
tff(cat_to_animal_type, type, cat_to_animal: cat > animal).
tff(dog_to_animal_type, type, dog_to_animal: dog > animal).
tff(garfield_type, type, garfield: cat).
tff(odie_type, type, odie: dog).
tff(jon_type, type, jon: human).
tff(owner_of_type, type, owner_of: animal > human).
tff(chased_type, type, chased: (dog * cat) > $o).
tff(hates_type, type, hates: (human * human) > $o).

tff(human_owner, axiom,
    ![A: animal]: ?[H: human]: H = owner_of(A)).

tff(jon_owns_garfield, axiom,
    jon = owner_of(cat_to_animal(garfield))).

tff(jon_owns_odie, axiom,
    jon = owner_of(dog_to_animal(odie))).

tff(jon_owns_only, axiom,
    ![A: animal]:
      (jon = owner_of(A)
     => (A = cat_to_animal(garfield)
       | A = dog_to_animal(odie)))).

tff(dog_chase_cat, axiom,
    ![C: cat, D: dog]:
      (chased(D, C)
     => hates(owner_of(cat_to_animal(C)),
              owner_of(dog_to_animal(D))))).

tff(odie_chased_garfield, axiom, chased(odie, garfield)).

tff(jon_hates_jon, conjecture, hates(jon, jon)).
```

Figure 6.1. TF0 Formulae.

- the predefined arithmetic types `$int`, `$rat`, and `$real`;
- user-defined n-ary type constructors applied to n type arguments; or
- type variables, which must be quantified by `!>` — see the type signature forms below.

Type constructors are declared before their use to be of the kind `($tType` $*\cdots*$ `$tType) > $tType`, in annotated formulae with a `type` role. Each TF1 type signature declares either

- an individual type τ;
- a function type $(\tau_1 * \cdots * \tau_n) > \tilde{\tau}$ for $n > 0$, where τ_i are the argument types and $\tilde{\tau}$ is the result type (with the same caveats as for TF0); or
- a polymorphic type `!>[`α_1`: $tType, ... ,`α_n`: $tType]:` ς for $n > 0$, where $\alpha_1, \ldots, \alpha_n$ are distinct type variables and ς is a TF0 type signature.

The `!>` binder in the last form denotes universal quantification in the style of $\lambda\Pi$ calculi. It is used only at the top level in polymorphic type signatures. All type variables must be of kind `$tType`; more complex type variables are beyond rank-1 polymorphism. An example of TF1 formulae can be found in [43].

6.2.2 FOOL

FOOL [50], standing for First-Order Logic (FOL) + bOoleans, is a variant of many-sorted first-order logic. FOOL extends FOL in that it (i) contains an interpreted boolean type, which allows boolean variables to be used as formulae, and allows all formulae to be used as boolean terms, (ii) contains conditional expressions, and (iii) contains let expressions. FOOL can be straightforwardly extended with the polymorphic theory of tuples that defines first class tuple types and terms [51]. In what follows we consider such extension, and tuples are part of TFX. There is a model-preserving transformation of FOOL formulae to FOL formulae [50] that can be implemented in a FOL ATP system to support reasoning with FOOL. Formulae of FOOL can also be efficiently translated to a first-order clausal normal form [49]. The following describes these features of FOOL, illustrating them using examples taken from [48] and [51]. The complete formal semantics of FOOL is given in [50].

Boolean Terms and Formulae

FOOL contains an interpreted two-element boolean type *bool*, allows quantification over variables of type *bool*, and considers formulae to be terms of type *bool*. This allows boolean variables to be used as formulae, and all formulae to be used as boolean terms. For example, Formula 6.1 is a syntactically correct tautology in FOOL.

$$(\forall x : bool)(x \vee \neg x) \tag{6.1}$$

Logical implication can be defined as a binary function *imply* of the type $bool \times bool \to bool$ using the axiom

$$(\forall x : bool)(\forall y : bool)(imply(x, y) \Leftrightarrow \neg x \vee y). \tag{6.2}$$

Then it is possible to express that P is a graph of a (partial) function of the type $\sigma \to \tau$ as

$$(\forall x : \sigma)(\forall y : \tau)(\forall z : \tau) imply(P(x, y) \wedge P(x, z), y \doteq z) \tag{6.3}$$

Formula 6.2 can be equivalently expressed with $\doteq$ instead of $\Leftrightarrow$.

Tuples

FOOL extended with the theory of tuples contains a type $(\sigma_1, \ldots, \sigma_n)$ of the n-ary tuple for all types $\sigma_1, \ldots, \sigma_n$, $n > 0$. Each type $(\sigma_1, \ldots, \sigma_n)$ is first class, that is, it can be used in the type of a function or predicate symbol, and in a quantifier. An expression $(t_1, \ldots, t_n)$, where $t_1, \ldots, t_n$ are terms of types $\sigma_1, \ldots, \sigma_n$, respectively, is a tuple term of type $(\sigma_1, \ldots, \sigma_n)$. Each tuple term is first class and can be used as an argument to a function symbol, a predicate symbol, or equality.

Tuples are ubiquitous in mathematics and programming languages. For example, one can use the tuple sort $(\mathbb{R}, \mathbb{R})$ as the sort of complex numbers. Thus the term $(2, 3)$ represents the complex number $2 + 3i$. A function symbol *plus* that represents addition of complex numbers has the type $(\mathbb{R}, \mathbb{R}) \times (\mathbb{R}, \mathbb{R}) \to (\mathbb{R}, \mathbb{R})$.

Conditional Expressions

FOOL contains conditional expressions of the form `if` ψ `then` s `else` t, where ψ is a formula, and s and t are terms of the same type. The semantics of such expressions mirrors the semantics of conditional expressions in programming languages, and they are therefore convenient for express-

ing formulae coming from program analysis. 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

$$(\forall x : \mathbb{Z})(\forall y : \mathbb{Z})(max(x, y) \doteq \texttt{if } x \geq y \texttt{ then } x \texttt{ else } y). \tag{6.4}$$

FOOL allows conditional expressions to occur as formulae, as in the following valid property of *max*.

$$(\forall x : \mathbb{Z})(\forall y : \mathbb{Z})(\texttt{if } max(x, y) \doteq x \texttt{ then } x \geq y \texttt{ else } y \geq x) \tag{6.5}$$

Let Expressions

FOOL contains let expressions of the form $\texttt{let } D_1; \ldots; D_k \texttt{ in } t$, where $k > 0$, t is either a term or a formula, and $D_1, \ldots, D_k$ are simultaneous non-recursive definitions. FOOL allows definitions of function symbols, predicate symbols, and tuples.

The definition of a function symbol $f : \sigma_1 \times \ldots \times \sigma_n \to \tau$ has the form $f(x_1 : \sigma_1, \ldots, x_n : \sigma_n) = s$, where $n \geq 0$, $x_1, \ldots, x_n$ are distinct variables, and s is a term of the type τ. For example, the following let expression denotes the maximum of three integer constants a, b, and c, using a local definition of the function symbol *max*.

$$\begin{array}{l} \texttt{let } max(x : \mathbb{Z}, y : \mathbb{Z}) = \texttt{if } x \geq y \texttt{ then } x \texttt{ else } y \\ \texttt{ in } max(max(a, b), c) \end{array} \tag{6.6}$$

The definition of a predicate symbol $p : \sigma_1 \times \ldots \times \sigma_n$ has the form $p(x_1 : \sigma_1, \ldots, x_n : \sigma_n) = \varphi$, where $n \geq 0$, $x_1, \ldots, x_n$ are distinct variables, and φ is a formula. For example, the following let expression denotes equivalence of two boolean constants A and B, using a local definition of the predicate symbol *imply*.

$$\begin{array}{l} \texttt{let } imply(x : bool, y : bool) = \neg x \vee y \\ \texttt{ in } imply(A, B) \wedge imply(B, A) \end{array} \tag{6.7}$$

The definition of a tuple has the form $(c_1, \ldots, c_n) = s$, where $n > 1$, $c_1, \ldots, c_n$ are distinct constant symbols of the types $\sigma_1, \ldots, \sigma_n$, respectively, and s is a term of the type $(\sigma_1, \ldots, \sigma_n)$. For example, the following formula defines addition for complex numbers using two simultaneous lo-

cal definition of tuples.

$$\begin{array}{l}(\forall x : (\mathbb{R}, \mathbb{R}))(\forall y : (\mathbb{R}, \mathbb{R})) \\ \quad (\mathit{plus}(x, y) \doteq \texttt{let } (a, b) = x;\ (c, d) = y \texttt{ in } (a + c, b + d))\end{array} \tag{6.8}$$

The semantics of let expressions in FOOL mirrors the semantics of simultaneous non-recursive local definitions in programming languages. That is, none of the definitions $D_1, \ldots, D_n$ uses function or predicate symbols created by any other definition. In the following example, constants a and b are swapped by a let expression. The resulting formula is equivalent to $P(b, a)$.

$$\texttt{let } a = b\,;\ b = a \texttt{ in } P(a, b) \tag{6.9}$$

Formula 6.9 can be equivalently expressed using the following let expression with a definition of a tuple.

$$\texttt{let } (a, b) = (b, a) \texttt{ in } P(a, b) \tag{6.10}$$

Let expressions with tuple definitions are convenient for expressing problems coming from program analysis, namely modelling of assignments [51]. The left hand side of Figure 6.2 shows an example of an imperative `if` statement containing assignments to integer variables, and an `assert` statement. This can be encoded in FOOL as shown on the right hand side, using let expressions with definitions of tuples that capture the assignments.

```
if (x > y) {
   t := x;
   x := y;
   y := t;
}
assert(x <= y);
```

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

Figure 6.2. FOOL encoding of an `if` statement.

6.3 The TFX Syntax

The TPTP TFF syntax has been extended to provide the features of FOOL, and at the same time some of the previous weaknesses have been remedied. Formulae and terms have been conflated (with some exceptions). Tuples have been removed from TFF, and fully expressive tuples included in TFX. The old conditional expressions and let expressions have been removed from TFF, and new elegant forms have been included as part of TFX. The grammar of TFX is captured in version v7.1.0.2 of the TPTP syntax, available online at http://www.tptp.org/TPTP/SyntaxBNF.html. In the subsections below, the relevant excerpts of the BNF are provided, with examples and commentary.

6.3.1 Boolean Terms and Formulae

Variables of type `$o` can be used as formulae, and formulae can be used as terms. The following is the relevant BNF excerpt. Formulae are terms are conflated by including logic/atomic formulae as options for terms/unitary terms. The distinction between formulae and terms is maintained for plain TFF.

```
<tff_logic_formula> ::=
    <tff_unitary_formula> | <tff_unary_formula>
  | <tff_binary_formula>  | <tff_defined_infix>
<tff_unitary_formula> ::=
    <tff_quantified_formula> | <tff_atomic_formula>
  | <tfx_unitary_formula>    | (<tff_logic_formula>)
<tfx_unitary_formula> ::= <variable>
<tff_term> ::=
    <tff_logic_formula> | <defined_term> | <tfx_tuple>
<tff_unitary_term> ::=
    <tff_atomic_formula> | <defined_term> | <tfx_tuple>
  | <variable> | (<tff_logic_formula>)
```

The FOOL tautology on Formula 6.1 can be written in TFX as

```
tff(tautology, conjecture, ![X: $o]: (X | ~X)).
```

The *imply* predicate in Formula 6.2 can be written in TFX as

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

The definition of a graph of a function on Formula 6.3 can be written in TFX as

```
tff(s, type, s: $tType).
tff(t, type, t: $tType).
tff(p, type, p: (s * t) > $o).
tff(graph, axiom,
    ![X: s, Y: t, Z: s]: imply(p(X, Y) & p(X, Z), Y = Z)).
```

A consequence of allowing formulae as terms is that the default typing of functions and predicates supported in plain TFF (functions default to `($i * ...* $i) > $i` and predicates default to `($i * ...* $i) > $o`) is not supported in TFX.

Note that not all terms can be used as formulae. Tuples, numbers, and "distinct objects" cannot be used as formulae.

6.3.2 Tuples

Tuples in TFX are written in `[]` brackets, and can contain any type of term, including formulae and variables of type `$o`. Signatures can contain tuple types. The following is the relevant BNF excerpt.

```
<tfx_tuple_type> ::= [<tff_type_list>]
<tff_type_list> ::=
    <tff_top_level_type>
  | <tff_top_level_type>,<tff_type_list>
<tfx_tuple> ::= [] | [<tff_arguments>]
<tff_arguments> ::= <tff_term> | <tff_term>,<tff_arguments>
```

The tuple type $(\mathbb{R}, \mathbb{R})$ can be written in TFX as `[$real, $real]` and the type of the addition function for complex numbers $(\mathbb{R}, \mathbb{R}) \times (\mathbb{R}, \mathbb{R}) \to (\mathbb{R}, \mathbb{R})$ can be written as

```
([$real, $real] * [$real, $real]) > [$real, $real].
```

The tuple term $(2, 3)$ can be written as `[2,3]`. Tuples can occur only as terms, anywhere they are well-typed (i.e., they cannot appear as formulae). In the following example the predicate `p` takes a tuple $(\mathbb{Z}, \iota, o)$ as the first argument.

```
tff(p_type, type, p: ([$int, $i, $o] * $o * $int) > $o).
tff(q_type, type, q: ($int * $i) > $o).
tff(me_type, type, me: $i).
tff(tuples_1, axiom,
    ![X: $int]: p([33, me, $true], ![Y: $i]: q(X, Y), 27)).
```

Note that while product types and tuple types are semantically equivalent, two separate syntaxes are used to make it easy to distinguish between the following cases.

```
tff(n_type, type, n: [$int, $int]).
tff(f_type, type, f: [$int, $int] > $int).
tff(g_type, type, g: ($int * $int) > $int).
tff(h_type, type, h: ([$int, $int] * $int) > $int).
```

The first case defines `n` to be a tuple of two integers. The second case defines `f` to be a function from a tuple of two integers to an integer. The third case defines `g` to be a function from two integers to an integer. The last case defines `h` to be a function from a tuple of two integers and an integer, to an integer.

The tuples syntax cannot be used to simultaneously declare types of multiple constants in an annotated formula with the `type` role. For example, the following expression is not valid.

```
tff(ab_type, type, [a, b]: [$int, $int]).
```

Instead, one must declare the type of each constant separately.

```
tff(a_type, type, a: $int).
tff(b_type, type, b: $int).
```

6.3.3 Conditional Expressions

Conditional expressions are polymorphic, taking a formula as the first argument, then two formulae or terms of the same type as the second and third arguments. The type of the conditional expression is the type of its second and third arguments. The following is the relevant BNF excerpt.

```
<tfx_conditional> ::= $ite(<tff_logic_formula>,
                           <tff_term>,<tff_term>)
```

The keyword `$ite` is used for conditional expressions occurring both as terms and formulae, which is different from the old TFF syntax of `if-then-else` that contained two separate keywords `$ite_t` and `$ite_f`.

The definition and a property of the *max* function on Formulae 6.4 and 6.5 can be expressed in TFX as

```
tff(max_type, type, max: ($int * $int) > $int).
tff(max_defn, axiom,
    ![X: $int, Y: $int]:
```

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

6.3.4 Let Expressions

Let expressions in TFX contain (i) the type signatures of locally defined symbols; (ii) the definitions of the symbols; and (iii) the term or formula in which the definitions are used. Type signatures in let expressions syntactically match those in annotated formulae with the `type` role. The symbol definitions determine how locally defined symbols are expanded in the term or formulae where they are used. The type signature must include the types for all the local defined symbols. The following is the relevant BNF excerpt.

```
<tfx_let> ::=
    $let(<tfx_let_types>,<tfx_let_defns>,<tff_term>)
<tfx_let_types> ::=
    <tff_atom_typing> | [<tff_atom_typing_list>]
<tff_atom_typing_list> ::=
    <tff_atom_typing>
  | <tff_atom_typing>,<tff_atom_typing_list>
<tfx_let_defns> ::= <tfx_let_defn> | [<tfx_let_defn_list>]
<tfx_let_defn> ::= <tfx_let_LHS> <assignment> <tff_term>
<tfx_let_LHS> ::= <tff_plain_atomic> | <tfx_tuple>
<tfx_let_defn_list> ::=
    <tfx_let_defn>
  | <tfx_let_defn>,<tfx_let_defn_list>
```

The keyword `$let` is used for let expressions defining both function and predicate symbols, regardless of whether the let expression occurs as a term or a formula. This is different from the old TFF syntax of let expressions that contained four separate keywords `$let_tt`, `$let_tf`, `$let_ft`, and `$let_ff`.

In the following example an integer constant `c` is defined in a let expression.

```
tff(p_type, type, p: ($int * $int) > $o).
tff(let_1, axiom, $let(c: $int, c:= $sum(2, 3), p(c, c))).
```

The left hand side of a definition may contain pairwise distinct variables as top-level arguments, which can also appear in the right hand side of the definition. Such variables are implicitly universally quantified, and of the type defined by the symbol's type declaration. The variables' values are supplied by unification in the defined symbol's use. Figures 6.3 and 6.4 show examples of let expressions with definitions of function and predicate symbols.

```
tff(max_max, axiom,
    $let(max: ($real * $real) > $real,
         max(X, Y) := $ite($greatereq(X, Y), X, Y),
         max(max(a, b), c))).
```

Figure 6.3. TFX encoding of Formula 6.6.

```
tff(a, type, a: $o).
tff(b, type, b: $o).
tff(a_eq_b, axiom,
    $let(imply: ($o * $o) > $o,
         imply(X, Y) := ~X | Y,
         imply(a, b) & imply(b, a))).
```

Figure 6.4. TFX encoding of Formula 6.7.

Let expression can use definitions of tuples. Formula 6.8 can be written in TFX as follows. Notice that the type declaration contains the elements of both tuples in the simultaneous definition.

```
tff(plus, type,
    plus: ([$real,$real] * [$real,$real]) > [$real,$real]).
tff(plus_def, axiom,
    ![X: [$real, $real], Y: [$real$, $real]]:
      (plus(X, Y)
      = $let([a: $real, b: $real, c: $real, d: $real],
             [[a, b] := X, [c, d] := Y],
             [$sum(a, c), $sum(b, d)])).
```

Sequential let expressions (`let*`) can be implemented by nesting. In the following example `ff` and `gg` are defined in sequence, and the let expression is equivalent to the formula `p(f(i,i,i,i))`.

```
tff(i_type, type, i: $int).
```

```
tff(f_type, type, f: ($int * $int * $int * $int) > $int).
tff(p_type, type, p: $int > $o).
tff(let_tuple_3, axiom,
    $let(ff: ($int * $int) > $int,
         ff(X, Y):= f(X, X, Y, Y),
         $let(gg: $int > $int,
              gg(Z) := ff(Z, Z),
              p(gg(i))))).
```

Let expressions can have simultaneous local definitions with the type declarations and the definitions given in `[]`es (they look like tuples of declarations and definitions, but are specified independently of tuples in the syntax). (Lisp-like programming languages call them `let`, and not `let*` — `let*` can be implemented in TFX by nesting `let`s). The symbols must have distinct signatures. Figure 6.5 shows two equivalent let expressions, one with a tuple definition, the other with two simultaneous definitions of constants.

```
tff(a, type, a: $i).
tff(b, type, b: $i).
tff(p, type, p: ($i*$i)>$o).
tff(pba, axiom,
    $let([a: $i, b: $i],
         [a := b, b := a],
         p(a, b))).
```

```
tff(a, type, a: $i).
tff(b, type, b: $i).
tff(p, type, p: ($i*$i)>$o).
tff(pba, axiom,
    $let([a: $i, b: $i],
         [a, b] := [b, a],
         p(a, b))).
```

Figure 6.5. TFX encodings of Formulas 6.9 (left) and 6.10 (right).

In the following example two function symbols are defined simultaneously, and the let expression is equivalent to the formula

```
p(f(i,i,f(i,i,i,i),f(i,i,i,i))).
```

```
tff(i_type, type, i: $int).
tff(f_type, type, f: ($int * $int * $int * $int) > $int).
tff(p_type, type, p: $int > $o).
tff(let_tuple_2, axiom,
    $let([ff: ($int * $int) > $int, gg: $int > $int],
         [ff(X, Y) := f(X,X,Y,Y), gg(Z) := f(Z,Z,Z,Z)],
         p(ff(i, gg(i))))).
```

The defined symbols of a let expression have scope over the formula/term in which the definitions are applied, shadowing any definition

outside the let expression. The right hand side of a definition can have symbols with the same name as the defined symbol, but refer to symbols defined outside the let expression. In the following example the local definition of the `array` function symbols shadow the global declaration.

```
tff(array_type, type, array: $int > $real).
tff(p_type, type, p: $real > $o).
tff(let_3, axiom,
    $let(array: $int > $real,
         array(I) := $ite(I = 3, 5.2, array(I)),
         p($sum(array(2), array(3))))).
```

6.4 Software Support and Examples

6.4.1 Software for TFX

The BNF provides the automatically generated lex/yacc parsers for TPTP files. At the time of writing this paper, the TPTP4X utility is being upgraded to support TFX.

The Vampire theorem prover [54] supports all features of FOOL. Vampire transforms FOOL formulae into a set of first-order clauses using the VCNF algorithm [49], and then reasons with these clauses using its usual resolution calculi for first-order logic. At the time of writing this paper the latest released version of Vampire, 4.2.2, uses a syntax for FOOL that slightly differs from TFX. Full support for the TFX syntax has been implemented in a recent revision of the Vampire source code[1], and will be available in the next release of Vampire.

TFX has been used by two program verification tools BLT [23] and Voogie [51]. Both BLT and Voogie read programs written in a subset of the Boogie intermediate verification language and generate their partial correctness properties written in the TFX syntax. BLT and Voogie generate formulae differently, but both rely on features of FOOL, namely conditional expressions, let expressions, and tuples.

6.4.2 Examples

Figures 6.6–6.8 show longer examples of useful applications of features of FOOL. Figure 6.6 shows how tuples, conditional expressions, and let expressions can be mixed, here to place two integer values in descending order as arguments in an atom. Figure 6.7 shows the TFX encoding of

[1] https://github.com/vprover/vampire

the FOOL formula in Figure 6.2, which expresses a partial correctness property of an imperative program with an `if` statement. Figure 6.8 shows an example that uses formulae as terms, in the second arguments of the `says` predicate. The problem is to find a model from which it is possible to determine which of `a`, `b`, or `c` is the only truthteller on this Smullyanesque island [77]. More TFX examples are available from the TPTP web site http://www.tptp.org/TPTP/Proposals/TFXExamples.tgz.

```
tff(v1_type, type, v1: $int).
tff(v2_type, type, v2: $int).
tff(ordered_p, axiom,
    $let([large: $int,small: $int],
         [large, small] := $ite($greater(v1,v2),
                                [v1, v2], [v2, v1]),
         p(large, small))).
```

Figure 6.6. Mixing tuples, conditional and let expressions.

```
tff(x, type, x: $int).
tff(y, type, y: $int).
tff(t, type, t: $int).
tff(x_leq_y, conjecture,
    $let([x: $int, y: $int, t: $int],
         [x, y, t] := $ite($greater(x,y),
                           $let(t: $int, t := x,
                           $let(x: $int, x := y,
                           $let(y: $int, y := t,
                                [x, y, t]))),
                           [x, y, t]),
         $lesseq(x, y))).
```

Figure 6.7. A TFX encoding of the program analysis problem in Figure 6.2.

6.5 Conclusion

This paper has introduced the eXtended Typed First-order form (TFX) of the TPTP's TFF language. TFX includes boolean variables as formulae, formulae as terms, tuple types and terms, conditional expressions, and

```
tff(a_type, type, a: $i).
tff(b_type, type, b: $i).
tff(c_type, type, c: $i).
tff(exactly_one_truthteller_type, type,
    exactly_one_truthteller: $o).
tff(says, type, says: ($i * $o) > $o).

% Each person is either a truthteller or a liar
tff(island, axiom,
    ![P: $i]: (says(P, $true) <~> says(P, $false))).
tff(exactly_one_truthteller, axiom,
    (exactly_one_truthteller
  <=> (?[P: $i]: says(P,$true)
      & ![P1: $i, P2: $i]:
          ((says(P1, $true) & says(P2, $true))
         => P1 = P2)))).

% B said that A said that there is
% exactly one truthteller on the island
tff(b_says, hypothesis,
    says(b, says(a, exactly_one_truthteller))).

% C said that what B said is false
tff(c_says, hypothesis, says(c, says(b, $false))).
```

Figure 6.8. Who is the truthteller?

let expressions. TFX is useful for (at least) concisely expressing problems coming from program analysis, and translated from more expressive logics.

Now that the syntax is settled, ATP system developers will be able to implement the new language features. It is already apparent from the SMT community that these are useful features, and systems that can already parse and reason using the SMT version 2 language need only new parsers to implement the features of TFX. In parallel, version v8.0.0 of the TPTP will include problems that use TFX, and the automated reasoning community is invited to submit problems for inclusion in TPTP.

Acknowledgements. Thanks to our friends in the TPTP World who have provided feedback on TFX features, starting from the TPTP Tea Party at CADE-22 in 2009. The second author was partially supported by the Wallenberg Academy Fellowship 2014 and the Swedish VR grant D0497701.

Bibliography

[1] Ehab Al-Shaer, Wilfredo Marrero, Adel El-Atawy, and Khalid Elbadawi. Network configuration in A box: Towards end-to-end verification of network reachability and security. In *Proceedings of the 17th annual IEEE International Conference on Network Protocols, 2009. ICNP 2009, Princeton, NJ, USA, 13-16 October 2009*, pages 123–132, 2009.
— One citation on page 114

[2] Noran Azmy and Christoph Weidenbach. Computing tiny clause normal forms. In *Automated Deduction – CADE-24*, pages 109–125. Springer, 2013.
— 2 citations on pages 6 and 68

[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 4

[4] Leo Bachmair and Harald Ganzinger. Resolution Theorem Proving. In *Handbook of Automated Reasoning*, pages 19–99. Elsevier and MIT Press, 2001.
— 3 citations on pages 4, 30, and 120

[5] Thomas Ball, Nikolaj Bjørner, Aaron Gember, Shachar Itzhaky, Aleksandr Karbyshev, Mooly Sagiv, Michael Schapira, and Asaf Valadarsky. VeriCon: towards verifying controller programs in software-defined networks. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'14, Edinburgh, UK — June 9–11, 2014*, pages 282–293, 2014.
— One citation on page 114

[6] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for

object-oriented programs. In *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005*, pages 364–387, 2005.
— 3 citations on pages 7, 105, and 111

[7] 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 7

[8] 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 18, 63, 85, and 120

[9] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.
— 2 citations on pages 68 and 89

[10] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories*, 2010.
— 10 citations on pages 11, 21, 32, 38, 62, 65, 84, 105, 127, and 131

[11] Clark W. Barrett, Leonardo Mendonça de Moura, and Aaron Stump. SMT-COMP: satisfiability modulo theories competition. In *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, pages 20–23, 2005.
— One citation on page 11

[12] Peter Baumgartner. SMTtoTPTP — A Converter for Theorem Proving Formats. In *Proceedings of CADE*, volume 9195 of *LNCS*, pages 285–294, 2015.
— 2 citations on pages 62 and 65

[13] Kamal Benzekki, Abdeslam El Fergougui, and Abdelbaki Elbelrhiti Elalaoui. Software-defined networking (sdn): a survey. *Security and communication networks*, 9(18):5803–5833, 2016.
— One citation on page 114

[14] 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 8 and 61

[15] Nikolaj Bjørner and Mikolas Janota. Playing with quantified satisfaction. In Ansgar Fehnker, Annabelle McIver, Geoff Sutcliffe, and Andrei Voronkov, editors, *LPAR-20. 20th International Conferences on Logic for Programming, Artificial Intelligence and Reasoning — Short Presentations*, volume 35 of *EPiC Series in Computing*, pages 15–27. EasyChair, 2015.
— One citation on page 90

[16] Nikolaj Bjørner and Karthick Jayaraman. Checking cloud contracts in microsoft azure. In *Distributed Computing and Internet Technology — 11th International Conference, ICDCIT 2015, Bhubaneswar, India, February 5-8, 2015. Proceedings*, pages 21–32, 2015.
— 2 citations on pages 114 and 126

[17] Jasmin Blanchette, Nicolas Peltier, and Simon Robillard. Superposition with Datatypes and Codatatypes. In D. Galmiche, S. Schulz, and R. Sebastiani, editors, *Proceedings of the 9th International Joint Conference on Automated Reasoning*, page To appear, 2018.
— One citation on page 7

[18] Jasmin Christian Blanchette and Andrei Paskevich. TFF1: The TPTP Typed First-order Form with Rank-1 Polymorphism. In M.P. Bonacina, editor, *Proceedings of the 24th International Conference on Automated Deduction*, number 7898 in Lecture Notes in Artificial Intelligence, pages 414–420. Springer-Verlag, 2013.
— 6 citations on pages 6, 32, 37, 65, 66, and 131

[19] Sascha Böhme and Tobias Nipkow. Sledgehammer: Judgement Day. In *Automated Reasoning, 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings*, pages 107–121, 2010.
— 2 citations on pages 8 and 20

[20] 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 7

[21] 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 8 and 61

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

[23] YuTing Chen and Carlo A. Furia. Triggerless happy – intermediate verification with a first-order prover. In Nadia Polikarpova and Steve Schneider, editors, *Proceedings of the 13th International Conference on integrated Formal Methods (iFM)*, volume 10510 of *Lecture Notes in Computer Science*, pages 295–311. Springer, September 2017.
— 4 citations on pages 105, 107, 111, and 145

[24] Koen Claessen and Niklas Sörensson. New techniques that improve mace-style finite model finding. In *Proceedings of the CADE-19 Workshop: Model Computation-Principles, Algorithms, Applications*, pages 11–27. Citeseer, 2003.
— One citation on page 120

[25] Keith L. Clark. Negation as failure. In *Logic and Data Bases, Symposium on Logic and Data Bases, Centre d'études et de recherches de Toulouse, 1977.*, pages 293–322, 1977.
— One citation on page 122

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

[27] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Proceedings of TACAS*, volume 4963 of *LNCS*, pages 337–340, 2008.
— 5 citations on pages 18, 63, 85, 107, and 126

[28] Nachum Dershowitz and David A. Plaisted. Rewriting. In *Handbook of Automated Reasoning (in 2 volumes)*, pages 535–610. 2001.
— One citation on page 4

[29] Ioan Drăgan and Laura Kovács. Lingva: Generating and Proving Program Properties Using Symbol Elimination. In *Proceedings of*

PSI, pages 67–75, 2014.
— One citation on page 19

[30] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin T. Vechev. Network-wide configuration synthesis. In *Computer Aided Verification — 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, pages 261–281, 2017.
— One citation on page 114

[31] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In *Programming Languages and Systems — 22nd European Symposium on Programming, ESOP 2013*, pages 125–128, 2013.
— 2 citations on pages 7 and 94

[32] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. A general approach to network configuration analysis. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 469–483, Berkeley, CA, USA, 2015. USENIX Association.
— One citation on page 114

[33] Nate Foster, Arjun Guha, Mark Reitblatt, Alec Story, Michael J. Freedman, Naga Praveen Katta, Christopher Monsanto, Joshua Reich, Jennifer Rexford, Cole Schlesinger, David Walker, and Rob Harrison. Languages for software-defined networks. *IEEE Communications Magazine*, 51(2):128–134, 2013.
— One citation on page 114

[34] Jean Goubault-Larrecq. Finite models for formal security proofs. *Journal of Computer Security*, 18(6):1247–1299, 2010.
— One citation on page 127

[35] Todd J. Green, Shan Shan Huang, Boon Thau Loo, and Wenchao Zhou. Datalog and recursive query processing. *Foundations and Trends in Databases*, 5(2):105–195, 2013.
— One citation on page 128

[36] Ashutosh Gupta, Laura Kovács, Bernhard Kragl, and Andrei Voronkov. Extensional crisis and proving identity. In *Automated Technology for Verification and Analysis — 12th International Symposium, ATVA 2014, Sydney, NSW, Australia, November 3–7, 2014,*

Proceedings, pages 185–200, 2014.
— 4 citations on pages 6, 18, 36, and 52

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

[38] 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.
— 3 citations on pages 33, 64, and 128

[39] Kryštof Hoder, Nikolaj Bjørner, and Leonardo De Moura. muZ: An efficient engine for fixed points with constraints. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 457–462, Berlin, Heidelberg, 2011. Springer-Verlag.
— One citation on page 114

[40] 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 7 and 17

[41] Kryštof Hoder and Andrei Voronkov. The 481 ways to split a clause and deal with propositional variables. In *Automated Deduction — CADE-24 — 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, pages 450–464, 2013.
— One citation on page 4

[42] Karthick Jayaraman, Nikolaj Bjørner, Geoff Outhred, and Charlie Kaufman. Automated analysis and debugging of network connectivity policies. *Microsoft Research*, pages 1–11, 2014.
— One citation on page 114

[43] Cezary Kaliszyk, Geoff Sutcliffe, and Fabian Rabe. TH1: The TPTP Typed Higher-Order Form with Rank-1 Polymorphism. In P. Fontaine, S. Schulz, and J. Urban, editors, *Proceedings of the 5th Workshop on the Practical Aspects of Automated Reasoning*, number 1635 in CEUR Workshop Proceedings, pages 41–55, 2016.
— 2 citations on pages 131 and 135

[44] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and Philip Brighten Godfrey. Veriflow: Verifying network-wide invariants

in real time. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*, pages 15–27, 2013.
— One citation on page 114

[45] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c: A software analysis perspective. *Formal Asp. Comput.*, 27(3):573–609, 2015.
— One citation on page 7

[46] 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 3 and 17

[47] Konstantin Korovin. Inst-gen — A modular approach to instantiation-based automated reasoning. In Andrei Voronkov and Christoph Weidenbach, editors, *Programming Logics — Essays in Memory of Harald Ganzinger*, volume 7797 of *Lecture Notes in Computer Science*, pages 239–270. Springer, 2013.
— 3 citations on pages 5, 90, and 128

[48] 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 and Proofs, 2016*, pages 37–48, 2016.
— 12 citations on pages 7, 13, 69, 79, 84, 88, 94, 102, 110, 111, 121, and 135

[49] Evgenii Kotelnikov, Laura Kovács, Martin Suda, and Andrei Voronkov. A clausal normal form translation for FOOL. In Christoph Benzmüller, Geoff Sutcliffe, and Raul Rojas, editors, *GCAI 2016. 2nd Global Conference on Artificial Intelligence*, volume 41 of *EPiC Series in Computing*, pages 53–71. EasyChair, 2016.
— 6 citations on pages 13, 94, 99, 110, 135, and 145

[50] Evgenii Kotelnikov, Laura Kovács, and Andrei Voronkov. A First Class Boolean Sort in First-Order Theorem Proving and TPTP. In *Intelligent Computer Mathematics — International Conference, CICM 2015, Washington, DC, USA, July 13–17, 2015, Proceedings*, pages 71–86, 2015.
— 18 citations on pages 12, 36, 38, 39, 40, 57, 64, 68, 70, 73, 74, 78, 84, 94, 103, 110, 131, and 135

[51] Evgenii Kotelnikov, Laura Kovács, and Andrei Voronkov. A FOOLish Encoding of the Next State Relations of Imperative Programs. In *Automated Reasoning — 9th International Joint Conference, IJCAR 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14–17, 2018, Proceedings*, Lecture Notes in Computer Science, pages 405–421. Springer, 2018.
— 6 citations on pages 14, 121, 131, 135, 138, and 145

[52] Laura Kovács, Simon Robillard, and Andrei Voronkov. Coming to terms with quantified reasoning. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017*, pages 260–270, 2017.
— One citation on page 99

[53] 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 7, 17, 50, and 58

[54] Laura Kovács and Andrei Voronkov. First-Order Theorem Proving and Vampire. In *Proceedings of CAV*, volume 8044 of *LNCS*, pages 1–35, 2013.
— 10 citations on pages 3, 17, 30, 36, 50, 69, 94, 105, 120, and 145

[55] Diego Kreutz, Fernando M. V. Ramos, Paulo Jorge Esteves Veríssimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, 2015.
— One citation on page 114

[56] K. Rustan M. Leino. Efficient weakest preconditions. *Inf. Process. Lett.*, 93(6):281–288, 2005.
— One citation on page 111

[57] K. Rustan M. Leino. This is Boogie 2. *Manuscript KRML*, 178(131), 2008.
— 3 citations on pages 94, 105, and 111

[58] John W. Lloyd. *Foundations of Logic Programming, 2nd Edition*. Springer, 1987.
— One citation on page 117

[59] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, Brighten Godfrey, and Samuel Talmadge King. Debugging the Data

Plane with Anteater. In *Proceedings of the ACM SIGCOMM 2011 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Toronto, ON, Canada, August 15-19, 2011*, pages 290–301, 2011.
— One citation on page 114

[60] William McCune. A Davis-Putnam program and its application to finite first-order model search: Quasigroup existence problems. Technical report, Argonne National Laboratory, 1994.
— One citation on page 120

[61] K. L. McMillan. Quantified Invariant Generation Using an Interpolating Saturation Prover. In *Proceedings of TACAS*, pages 413–427, 2008.
— 2 citations on pages 7 and 17

[62] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *J. ACM*, 53(6):937–977, 2006.
— One citation on page 90

[63] 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.
— 3 citations on pages 4, 30, and 120

[64] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. 2002.
— 4 citations on pages 7, 20, 38, and 61

[65] Andreas Nonnengart and Christoph Weidenbach. Computing small clause normal forms. *Handbook of Automated Reasoning*, 1:335–367, 2001.
— 3 citations on pages 6, 68, and 69

[66] Giles Reger, Nikolaj Bjørner, Martin Suda, and Andrei Voronkov. AVATAR modulo theories. In *GCAI 2016. 2nd Global Conference on Artificial Intelligence*, pages 39–52, 2016.
— 3 citations on pages 6, 108, and 120

[67] Giles Reger, Martin Suda, and Andrei Voronkov. VCNF: a new clausification algorithm for first-order logic. In preparation.
— 4 citations on pages 6, 69, 70, and 71

[68] Giles Reger, Martin Suda, and Andrei Voronkov. The Challenges of Evaluating a New Feature in Vampire. In *Proceedings of the 1st and 2nd Vampire Workshops, Vampire@VSL 2014, Vienna, Austria, July 23, 2014 / Vampire@CADE 2015, Berlin, Germany, August 2, 2015*, pages 70–74, 2014.
— One citation on page 127

[69] Giles Reger, Martin Suda, and Andrei Voronkov. Finding finite models in multi-sorted first-order logic. In *Theory and Applications of Satisfiability Testing — SAT 2016 – 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, pages 323–341, 2016.
— 2 citations on pages 5 and 120

[70] 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.
— 3 citations on pages 62, 84, and 85

[71] Andrew Reynolds, Tim King, and Viktor Kuncak. An instantiation-based approach for solving quantified linear arithmetic. *CoRR*, abs/1510.02642, 2015.
— One citation on page 90

[72] Andrew Reynolds, Cesare Tinelli, Amit Goel, and Sava Krstić. Finite model finding in SMT. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, pages 640–655, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
— One citation on page 120

[73] 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 6 and 31

[74] John Alan Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.
— One citation on page 4

[75] Stephan Schulz. System Description: E 1.8. In *Proceedings of LPAR*, volume 8312 of *LNCS*, pages 735–743, 2013.
— 5 citations on pages 3, 17, 30, 61, and 120

[76] Martina Seidl, Florian Lonsing, and Armin Biere. qbf2epr: A tool for generating EPR formulas from QBF. In Pascal Fontaine, Renate A. Schmidt, and Stephan Schulz, editors, *Third Workshop on*

Practical Aspects of Automated Reasoning, PAAR-2012, Manchester, UK, June 30 – July 1, 2012, volume 21 of *EPiC Series in Computing*, pages 139–148. EasyChair, 2012.
— One citation on page 90

[77] Raymond M. Smullyan. *What is the Name of This Book? The Riddle of Dracula and Other Logical Puzzles*. Prentice-Hall, 1978.
— One citation on page 146

[78] 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.
— 2 citations on pages 87 and 109

[79] Geoff Sutcliffe. The TPTP Problem Library and Associated Infrastructure. *Journal of Automated Reasoning*, 43(4):337–362, 2009.
— 10 citations on pages 6, 18, 32, 36, 37, 61, 85, 107, 121, and 127

[80] Geoff Sutcliffe. The TPTP Problem Library and Associated Infrastructure. The FOF and CNF Parts, v3.5.0. *Journal of Automated Reasoning*, 43(4):337–362, 2009.
— One citation on page 131

[81] Geoff Sutcliffe. The TPTP World — Infrastructure for Automated Reasoning. In E. Clarke and A. Voronkov, editors, *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, number 6355 in Lecture Notes in Artificial Intelligence, pages 1–12. Springer-Verlag, 2010.
— One citation on page 131

[82] 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 61

[83] Geoff Sutcliffe. The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0. *Journal of Automated Reasoning*, 59(4):483–502, 2017.
— One citation on page 131

[84] 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.
— 2 citations on pages 44 and 131

[85] Geoff Sutcliffe and Evgenii Kotelnikov. TFX: The TPTP Extended Typed First-order Form. In *Proceedings of the 6th workshop on Practical Aspects of Automated Reasoning*, page To appear, 2018.
— One citation on page 15

[86] Geoff Sutcliffe, Stephan Schulz, Koen Claessen, and Peter Baumgartner. The TPTP Typed First-order Form with Arithmetic. In N. Bjørner and A. Voronkov, editors, *Proceedings of the 18th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, number 7180 in Lecture Notes in Artificial Intelligence, pages 406–419. Springer-Verlag, 2012.
— 5 citations on pages 6, 36, 99, 131, and 133

[87] Geoff Sutcliffe and Christian Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.
— One citation on page 131

[88] Geoff Sutcliffe and Christian Suttner. The State of CASC. *AI Communications*, 19(1):35–48, 2006.
— One citation on page 6

[89] Tanel Tammet. Gandalf. *Journal of Automated Reasoning*, 18(2):199–204, 1997.
— One citation on page 120

[90] Andrzej Trybulec. Mizar. In *The Seventeen Provers of the World, Foreword by Dana S. Scott*, pages 20–23, 2006.
— 2 citations on pages 7 and 20

[91] G.S. Tseitin. On the complexity of derivation in propositional calculus. In Jörg H. Siekmann and Graham Wrightson, editors, *Automation of Reasoning*, Symbolic Computation, pages 466–483. Springer Berlin Heidelberg, 1983.
— One citation on page 90

[92] 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 8 and 20

[93] Andrei Voronkov. AVATAR: the architecture for first-order theorem provers. In *Computer Aided Verification — 26th International Conference, CAV 2014*, pages 696–710, 2014.
— 2 citations on pages 4 and 108

[94] Christoph Weidenbach. Towards an automatic analysis of security protocols in first-order logic. In *Automated Deduction — CADE-16, 16th International Conference on Automated Deduction, Trento, Italy, July 7-10, 1999, Proceedings*, pages 314–328, 1999.
— One citation on page 127

[95] Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. Spass version 3.5. In *CADE*, pages 140–145, 2009.
— 2 citations on pages 30 and 120

[96] Max Wisniewski, Alexander Steen, Kim Kern, and Christoph Benzmüller. Effective normalization techniques for HOL. In Nicola Olivetti and Ashish Tiwari, editors, *Automated Reasoning — 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27 – July 2, 2016, Proceedings*, volume 9706 of *Lecture Notes in Computer Science*, pages 362–370. Springer, 2016.
— One citation on page 89

[97] 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 6