The Vampire and the FOOL

Evgenii Kotelnikov
Chalmers University of Technology, Gothenburg, Sweden
evgenyk@chalmers.se

Laura Kovács
Chalmers University of Technology, Gothenburg, Sweden
laura.kovacs@chalmers.se

Giles Reger
The University of Manchester, Manchester, UK
giles.reger@manchester.ac.uk

Andrei Voronkov
The University of Manchester; Chalmers University of Technology; EasyChair
andrei@voronkov.com

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.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; F.4.1 [*Mathematical Logic and Formal Languages*]: Specifying and Verifying and Reasoning about Programs; F.4.1 [*Artificial Intelligence*]: Deduction and Theorem Proving; F.4.1 [*Artificial Intelligence*]: Programming Languages and Software

Keywords automated theorem proving, first-order logic, program analysis, program verification, Vampire, TPTP

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, [8] shows a considerable gain in performance on proving properties of data collections by using a specially designed extensionality resolution rule.

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

In this work we present new features recently developed and implemented in the theorem prover Vampire [13] to natively support mirroring programming language constructs in its input language. They include (i) FOOL [10], 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 presents how FOOL is implemented in Vampire and focuses on new extensions to the TPTP input language [17] of first-order provers. Section 2 extends the TPTP language of monomorphic many-sorted first-order formulas, called TFF0 [20], and allows users to treat the built-in boolean sort `$o` as a first class sort. Moreover, it introduces expressions `$ite` and `$let` , which unify various TPTP `if-then-else` and `let-in` expressions.

Section 3 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 [5] of rank-1 polymorphic many-sorted 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 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 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 6 discusses related work and Section 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 [17], 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 [14]. 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. First Class Boolean Sort

Our recent work [10] 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 (1) treating boolean terms as formulas; (2) `if-then-else` expressions; and (3) `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 [1], 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.

2.1 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 [10], 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 [10] 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. They are controlled by the option `--fool_paramodulation`, which can be set to `on` or `off`. 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 [10]. 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` options 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 Quantifiers over the Boolean Sort

FOOL allows quantification over *bool* and usage of boolean variables as formulas. For example, the formula $(\forall x : \mathit{bool})(x \vee \neg x)$ 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 $x \doteq \mathit{true}$. For example, the tautology $(\forall x : \mathit{bool})(x \vee \neg x)$ will be converted to the FOL formula $(\forall x : \mathit{bool})(x \doteq \mathit{true} \vee x \not\doteq \mathit{true})$ 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 $(\forall x : \mathit{bool})(x \vee \neg x)$ can be expressed in this syntax as `![X:$o]: (X | ~X).`

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 $\mathit{bool} \times \mathit{bool} \to \mathit{bool}$ using the following axiom:

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

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

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

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

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:

$$(\forall x : \sigma)(\forall y : \tau)(\forall z : \tau)\mathit{impl}(P(x, y) \wedge P(x, z), y \doteq z). \quad (1)$$

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
$$\sigma_1 \times \ldots \times \sigma_n \to \mathit{bool};$$
2. adds the definition
$$(\forall x_1 : \sigma_1) \ldots (\forall x_n : \sigma_n)(\psi \Leftrightarrow g(x_1, \ldots, x_n) \doteq \mathit{true})$$
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*, (1) becomes

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

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

1. $(\forall x : \sigma)(\forall y : \tau)(\forall z : \tau)(P(x, y) \wedge P(x, z) \Leftrightarrow g_1(x, y, z) \doteq \mathit{true})$;
2. $(\forall y : \tau)(\forall z : \tau)(y \doteq z \Leftrightarrow g_2(y, z) \doteq \mathit{true})$.

TFF0 does not allow formulas to occur as arguments. The extended version of TFF0, supported by Vampire, removes this restriction for arguments of the boolean sort. Formula (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) [19]. 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.

2.5 `if-then-else`

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

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

$$(\forall x : \mathrm{Z})(\forall y : \mathrm{Z})(max(x, y) \doteq \texttt{if } x \geq y \texttt{ then } x \texttt{ else } y). \quad (2)$$

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

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

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

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

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*:

$$(\forall x : \mathrm{Z})(\forall y : \mathrm{Z})(\texttt{if } max(x,y) \doteq x \texttt{ then } x \geq y \texttt{ else } y \geq x). \tag{3}$$

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) and `$ite_f` in (3).

2.6 let-in

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

$$\begin{array}{l}\texttt{let } f_1(x_1^1 : \sigma_1^1, \ldots, x_{n_1}^1 : \sigma_{n_1}^1) = s_1; \\ \quad \ldots \\ \quad f_m(x_1^m : \sigma_1^m, \ldots, x_{n_m}^m : \sigma_{n_m}^m) = s_m \\ \texttt{in } t,\end{array} \tag{4}$$

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

$$\begin{array}{l}\texttt{let } f_1(x_1^1 : \sigma_1^1, \ldots, x_{n_1}^1 : \sigma_{n_1}^1) = s_1 \texttt{ in} \\ \quad \ddots \\ \quad \texttt{let } f_m(x_1^m : \sigma_1^m, \ldots, x_{n_m}^m : \sigma_{n_m}^m) = s_m \texttt{ in} \\ \quad\quad t.\end{array} \tag{5}$$

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 $\mathrm{Z} \to \mathrm{Z}$ 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`.

$$\begin{array}{l}\texttt{let } array(i : \mathrm{Z}) = \texttt{if } i \doteq 3 \texttt{ then } 5 \texttt{ else } array(i) \texttt{ in} \\ \quad array(2) + array(3)\end{array} \tag{6}$$

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 $f(b, a)$.

$$\texttt{let } a = b;\ b = a \texttt{ in } f(a, b) \tag{7}$$

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

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

      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 (6) becomes $g(2) + g(3)$, where g is a fresh function symbol of the type $\mathrm{Z} \to \mathrm{Z}$ defined by the following formula:

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

The example (7) is translated in the following way. First, the `let-in` expression is translated to the form (5). The constant a has a free occurrence in the body of b, therefore it is replaced by a fresh constant a'. The formula (7) becomes

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

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

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

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

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

Formula (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 `$let` expression can be used for multiple bindings. For that, the bindings should be separated by a semicolon and passed as the first argument. The formula (7) can be written using `$let` as follows.

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

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

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

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

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

3. 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 [12]. Previous versions of Vampire supported theories of integer arrays and arrays of integer arrays [13]. No other array sorts were supported and in order to implement one it would be necessary to hardcode a new sort and add the theory axioms corresponding to that sort. In this section we describe a polymorphic theory of arrays implemented in Vampire.

3.1 Definition

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

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

1. read-over-write 1
$$(\forall a : array(\tau, \sigma))(\forall v : \sigma)(\forall i : \tau)(\forall j : \tau) \\ (i \doteq j \Rightarrow select(store(a, i, v), j) \doteq v);$$
2. read-over-write 2
$$(\forall a : array(\tau, \sigma))(\forall v : \sigma)(\forall i : \tau)(\forall j : \tau) \\ (i \not\doteq j \Rightarrow select(store(a, i, v), j) \doteq select(a, j));$$
3. extensionality
$$(\forall a : array(\tau, \sigma))(\forall b : array(\tau, \sigma)) \\ ((\forall i : \tau)(select(a, i) \doteq select(b, i)) \Rightarrow a \doteq b).$$

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

One can use the polymorphic theory of arrays to express program properties. Recall the code snippet involving arrays mentioned in Section 2:

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

Formula (6) used an interpreted function to represent the array in this code. We can alternatively use arrays to repre-

sent it as follows

$$\begin{array}{l}\texttt{let } array = store(array, 3, 5) \texttt{ in} \\ \quad select(array, 2) + select(array, 3)\end{array} \tag{8}$$

3.2 Implementation in Vampire

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

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

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

$$\begin{array}{l}(\forall a : array(\tau, \sigma))(\forall i : \tau)(\forall v : \sigma) \\ \quad (select(store(a, i, v), i) \doteq v)\end{array} \tag{9}$$

$$\begin{array}{l}(\forall a : array(\tau, \sigma))(\forall i : \tau)(\forall j : \tau)(\forall v : \sigma) \\ \quad (i \not\doteq j \Rightarrow select(store(a, i, v), j) \doteq select(a, j))\end{array} \tag{10}$$

$$\begin{array}{l}(\forall a : array(\tau, \sigma))(\forall b : array(\tau, \sigma)) \\ \quad (a \not\doteq b \Rightarrow (\exists i : \tau)(select(a, i) \not\doteq select(b, i)))\end{array} \tag{11}$$

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 (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 [8] to efficiently reason with the extensionality axiom.

To express arrays, the TPTP syntax extension supported by Vampire allows, for every pair of sorts τ and σ, denoted by `t` and `s` in the TFF0 syntax, to denote the sort $array(\tau, \sigma)$ by `$array(s,t)`. Function symbols *select* and *store* can be expressed as ad-hoc polymorphic `$select` and `$store`, respectively for every pairs of sorts τ, σ. Previously, the theories of integer arrays and arrays of integer arrays were represented as sorts `$array1` and `$array2` in Vampire, with the corresponding sort-specific function symbols `$select1`, `$select2`, `$store1` and `$store2`. Our new implementation in Vampire, with support for the polymorphic theory of arrays, deprecates these two concrete array theories. Instead, one can now use the sorts `$array($int,$int)` and `$array ($int,$array($int,$int))`. For example, formula (8) can be written in the extended TFF0 syntax as follows:

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

3.3 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:

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

$$\begin{array}{l}(\forall a : array(\tau, bool))(\forall i : \tau)(\forall j : \tau)(\forall v : bool) \\ \quad (i \not\doteq j \Rightarrow select(store(a, i, v), j) \Leftrightarrow select(a, j))\end{array}$$

$$\begin{array}{l}(\forall a : array(\tau, bool))(\forall b : array(\tau, bool)) \\ \quad (a \not\doteq b \Rightarrow (\exists i : \tau)(select(a, i) \oplus select(b, i)))\end{array}$$

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 $array(\mathrm{Z}, bool) \times array(\mathrm{Z}, bool) \to array(\mathrm{Z}, bool)$. 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:

$$\begin{array}{l}(\forall message : array(\mathrm{Z}, bool))(\forall key : array(\mathrm{Z}, bool))(\forall i : \mathrm{Z}) \\ \quad (select(encrypt(message, key), i) \doteq \\ \quad\quad select(message, i) \oplus select(key, i)).\end{array}$$

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.

$$\begin{array}{l}(\forall plaintext : array(\mathrm{Z}, bool))(\forall cipher : array(\mathrm{Z}, bool)) \\ \quad (\forall key : array(\mathrm{Z}, bool))(cipher \doteq encrypt(plaintext, key) \Rightarrow \\ \quad\quad key \doteq encrypt(plaintext, cipher))\end{array}$$

The sort $array(\mathrm{Z}, bool)$ 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)))).
```

```
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 1. Sequence of conditionals.

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

Figure 2. Updating multiple variables.

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

invariant a=b+c ∧
a≥ 0∧ b≥ 0 ∧ c≥ 0 ∧ a≤ k ∧
$(\forall p)(0 \leq p < b \Rightarrow (\exists i)(0 \leq i < a \wedge A[i] > 0 \wedge B[p] = A[i]))$

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

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

Figure 3. Array partition.

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

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

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

Figure 4. Representation of the partial correctness statement of Figure 1.

4. Program Analysis with the New Extensions

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

4.1 Encoding the next state relation

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

Figure 4 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 1 in FOOL. Note that each expression from the program appears only once in the encoding.

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

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

Given a program P, the next state value for variable x can be given by [P; x := 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 that ensures that

```
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 5. Representation of the partial correctness statement of Figure 3 in Vampire.

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 can be encoded as follows:

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

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 [10]. 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:

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

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 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.

4.2 A program with a loop and arrays

Let us now show the use of FOOL in Vampire for reasoning about programs with loops. Consider the program given in Figure 3, written in a C-like syntax. The program fills an integer-valued array B by the strictly positive values of a source array A, and an integer-valued array C with the non-positive values of A. A safety assertion, in FOL, is specified at the end of the loop, using the **assert** construct. The program of Figure 3 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 [12]. In this case we use the FOL property specified in the **invariant** construct of Figure 3. This invariant property states that at any loop iteration, (i) the sum of visited array elements in A is the sum of visited elements in B and C (that is, $a = b + c$), (ii) the number of visited array elements in A, B, C is positive (that is, $a \geq 0$, $b \geq 0$, and $c \geq 0$), with $a \leq k$, and (iii) each array element $B[0], \ldots, B[b-1]$ is a strictly positive element in A. Formulating the latter property requires quantifier alternation in FOL, resulting in the quantified property with $\forall\exists$ listed in the invariant of Figure 3. 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 5 shows the encoding in the extended TFF0 syntax of this partial correctness statement; note that this uses the built-in theory of polymorphic arrays in Vampire, where $arrayA$, $arrayB$ and $arrayC$ correspond respectively to the arrays A, B and C.

So far, we assumed that the given invariant in Figure 3 is indeed an invariant. Using FOOL+ described in Section 4.1, we can verify the inductiveness property of the invariant, as follows: (i) express the the transition relation of the loop in FOOL+, and (ii) prove that, if the invariant holds at an arbitrary loop iteration i, then it also holds at loop iteration i+1. For proving this, we can again use FOOL+ to formulate the next state values of loop variables in the invariant at loop iteration i+1. Moreover, FOOL+ can also be used to express formulas as inputs to the symbol elimination method for invariant generation in Vampire. We leave the task of using FOOL+ for invariant generation as further work.

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 $\star$ 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 [17] 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 [14], see Subsection 5.2 for more details.

Our results are summarised in Tables 1–3 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/`.

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)`.

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

Table 1. Runtimes in seconds of provers on the set of 134 higher-order TPTP problems.

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

For this experiment, we compared the performance of Vampire with those of the higher-order theorem provers used in the the latest edition of CASC [18]: Satallax [6], Leo-II [4], and Isabelle [14]. We note that all of them used the first-order theorem prover E [16] for first-order reasoning (Isabelle also used several other provers).

Table 1 summarises our results on these problems. Only Vampire, Vampire $\star$ 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, these problems previously required higher-order logic, but now can be proven using first-order reasoning. Moreover, even on such simple problems there is a clear gain from using FOOL paramodulation.

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 [1]. 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 Jasmine Blanchette, following the recent work on reasoning about (co)datatypes [15]. To run the benchmark we first translated the SMT-LIB files to the TPTP syntax using the SMTtoTPTP translator [3] version 0.9.2. Let us note that this version of SMTtoTPTP does not fully support the boolean type in SMT-LIB. However, by setting the option `--keepBool` in SMTtoTPTP, we managed to translate these 152 problems into an extension of TFF0, which Vampire can read. We also modified the source code of SMTtoTPTP so that `if-then-else` expressions in the SMT-LIB files are not expanded but translated to `$ite` in FOOL. A similar modification would have been needed for translating `let-in` expressions; however, none of our 152 examples used `let-in`.

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

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

Table 2. Runtimes in seconds of provers on the set of 152 algebraic datatypes problems.

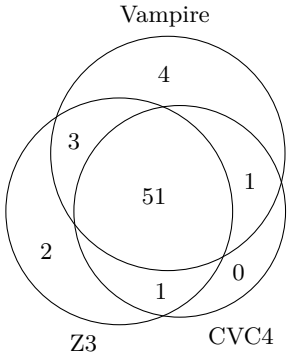

Figure 6. Venn diagram of the subsets of the algebraic datatypes problems, solved by Vampire, CVC4 and Z3.

compared Vampire with the best available SMT solvers, namely with CVC4 [2] and Z3 [7].

Table 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 6 shows the Venn diagram of the sets of problems solved by Vampire, CVC4 and Z3, where the numbers denote the numbers of solved problems. All problems apart from 11 were either solved by all systems or not solved by all systems. Table 3 details performance results on these 11 problems.

Based on our experimental results shown in Tables 2 and 3, we make the following observations. On the given set of problems the implementation of FOOL reasoning in Vampire was efficient enough to compete with state-of-the-art SMT solvers. This is significant because the problems were tailored for SMT reasoning. Vampire not only solved the largest number of problems, but also yielded runtime results that are comparable with those of CVC4. Whenever successful, Z3 turned out to be faster than Vampire; we believe this is because of the sophisticated 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 on the number of solved problems.

6. Related Work

FOOL was introduced in our previous work [10]. 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 [9]. The approach of [9] 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 [9] 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 [1] 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 [3] 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 [5].

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 give a simple extension to FOOL, allowing 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

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

Table 3. Runtimes in seconds of provers on selected algebraic datatypes problems. Dashes mean the solver failed to find a solution.

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 [5] syntax. Note that the current usage of `$array` conforms with the TFF1 syntax for type constructors.

References

[1] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0. Technical report, Department of Computer Science, The University of Iowa, 2010. Available at `www.SMT-LIB.org`.

[2] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. CVC4. In *Proc. of CAV*, volume 6806 of *LNCS*, pages 171–177, 2011.

[3] P. Baumgartner. SMTtoTPTP — a converter for theorem proving formats. In *Proc. of CADE*, volume 9195 of *LNCS*, pages 285–294, 2015.

[4] C. Benzmüller, L. Paulson, F. Theiss, and A. Fietzke. LEO-II - A Cooperative Automatic Theorem Prover for Higher-Order Logic. In *Proc. of IJCAR*, volume 5195 of *LNAI*, pages 162–170, 2008.

[5] J. C. Blanchette and A. Paskevich. TFF1: The TPTP Typed First-Order Form with Rank-1 Polymorphism. In *Proc. of CADE*, volume 7898 of *LNCS*, pages 414–420, 2013.

[6] C. Brown. Satallax: An Automated Higher-Order Prover (System Description). In *Proc. of IJCAR*, volume 7364 of *LNAI*, pages 111–117, 2012.

[7] L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In *Proc. of TACAS*, volume 4963 of *LNCS*, pages 337–340, 2008.

[8] A. Gupta, L. Kovács, B. Kragl, and A. Voronkov. Extensionality Crisis and Proving Identity. In *Proc. of ATVA*, pages 185–200, 2014.

[9] T. Hillenbrand and C. Weidenbach. Superposition for Bounded Domains. In *Automated Reasoning and Mathematics - Essays in Memory of William W. McCune*, pages 68–100, 2013.

[10] E. Kotelnikov, L. Kovács, and A. Voronkov. A First Class Boolean Sort in First-Order Theorem Proving and TPTP. In *Proc. of CICM*, volume 9150 of *LNCS*, pages 71–86, 2015.

[11] L. Kovács and A. Voronkov. Finding Loop Invariants for Programs over Arrays Using a Theorem Prover. In *Proc. of FASE*, pages 470–485, 2009.

[12] L. Kovács and A. Voronkov. Finding Loop Invariants for Programs over Arrays Using a Theorem Prover. In *Proc. of FASE*, volume 5503 of *LNCS*, pages 470–485, 2009. ISBN 978-3-642-00592-3.

[13] L. Kovács and A. Voronkov. First-Order Theorem Proving and Vampire. In *Proc. of CAV*, volume 8044 of *LNCS*, pages 1–35, 2013. ISBN 978-3-642-39798-1.

[14] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[15] A. Reynolds and J. Blanchette. A Decision Procedure for (Co)datatypes in SMT Solvers. In *Proc. of CADE*, volume 9195 of *LNCS*, pages 197–213, 2015.

[16] S. Schulz. System Description: E 1.8. In *Proc. of LPAR*, volume 8312 of *LNCS*, pages 735–743, 2013.

[17] G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure. *J. Autom. Reasoning*, 43(4):337–362, 2009.

[18] G. 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.

[19] G. Sutcliffe and C. Benzmüller. Automated reasoning in higher-order logic using the TPTP THF infrastructure. *J. Formalized Reasoning*, 3(1):1–27, 2010. .

[20] G. Sutcliffe, S. Schulz, K. Claessen, and P. Baumgartner. The TPTP Typed First-Order Form with Arithmetic. In *Proc. of LPAR*, volume 7180 of *LNCS*, pages 406–419, 2012.