EasyChair Preprint

№ 98

A FOOLish Encoding of the Next State Relations of Imperative Programs

Evgenii Kotelnikov, Laura Kovács and Andrei Voronkov

April 25, 2018

A FOOLish Encoding of the Next State Relations of Imperative Programs

Evgenii Kotelnikov[1], Laura Kovács[1,2], and Andrei Voronkov[3,4]

[1] Chalmers University of Technology, Gothenburg, Sweden
[2] TU Wien, Vienna, Austria
[3] The University of Manchester, Manchester, UK
[4] EasyChair

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.

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 [12] and WhyML [5], 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 [8], 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 [10] to support FOOL [6] and designed an efficient clausification algorithm VCNF [7] 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 $\texttt{if}\ \varphi\ \texttt{then}\ s\ \texttt{else}\ t$, where φ is a formula, and s and t are either both terms or formulas.
- 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 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) $\texttt{if}\ \varphi\ \texttt{then}\ s_1\ \texttt{else}\ s_2$, where φ is a formula, and s_1 and s_2 are tuple expressions; or
     (c) $\texttt{let}\ D_1; \ldots; D_k\ \texttt{in}\ s'$, 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 1. This program contains

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

Fig. 1. An imperative program with an `if` statement.

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

Fig. 2. A FOOL encoding of the program assertion on Figure 1.

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. 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 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 1 is concisely expressed with our encoding as the FOOL formula on Figure 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 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). 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 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 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). 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.

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

$$\begin{aligned}&(\forall x_1 : \tau_1) \ldots (\forall x_n : \tau_n)(\forall y_1 : \tau_1) \ldots (\forall y_n : \tau_n)\\ &\quad((x_1, \ldots, x_n) \doteq (y_1, \ldots, y_n) \Rightarrow x_1 \doteq y_1 \wedge \ldots \wedge x_n \doteq y_n).\end{aligned}$$

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

$$\begin{aligned}&(\forall x : (\mathbb{R}, \mathbb{R}))(\forall y : (\mathbb{R}, \mathbb{R}))\\ &\quad(plus(x, y) \doteq (\pi_1(x) + \pi_1(y), \pi_2(x) + \pi_2(y))),\end{aligned} \tag{1}$$

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 1 can be equivalently expressed using a `let-in` with two simultaneous tuple definitions as follows

$$\begin{aligned}&(\forall x : (\mathbb{R}, \mathbb{R}))(\forall y : (\mathbb{R}, \mathbb{R}))\\ &\quad(plus(x, y) \doteq \texttt{let}\ (a, b) = x;\ (c, d) = y\ \texttt{in}\ (a + c, b + d)).\end{aligned} \tag{2}$$

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 [9] 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 [16]. The TFX subset[5] 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 as `[$sum(a,c),$sum(b,d)]`. Formula 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)]))).
```

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

Vampire translates `let-in` with tuple definitions to clausal normal form of first-order logic using the VCNF clausification algorithm [7].

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 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 of this encoding in Section 3.2. Finally, in Section 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 6) and soundness of this encoding (Theorem 1).

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 multidimentional arrays. We do not introduce a distinguished type for multidimentional 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 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 σ_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 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* $x_1, \ldots, x_n \coloneqq e_1, \ldots, e_n$ *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* $x[e_1, \ldots, e_n] \coloneqq e$ *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 $x_1, \ldots, x_n \coloneqq e_1, \ldots, e_n$ and x in the statement $x[e_1, \ldots, e_n] \coloneqq e$ 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 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. *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* $[\![s_2]\!] \circ [\![s_1]\!]$*.* □

3.2 Encoding the Next State Relation

Our setting is FOOL extended with the theory of linear integer arithmetic, the polymorphic theory of arrays [6], and the polymorphic theory of first class tuples (Section 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 multidimentional arrays in FOOL as nested arrays[6]. To this end we (i) inductively define $\mathit{select}(a, i_1, \ldots, i_n)$, where $n > 1$, to be $\mathit{select}(\mathit{select}(a, i_1), i_2, \ldots, i_n)$; and (ii) inductively define $\mathit{store}(a, i_1, \ldots, i_n, e)$, where $n > 1$, to be $\mathit{store}(a, i_1, \mathit{store}(\mathit{select}(a, i_1), i_2, \ldots, i_n, e))$.

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 [8] for the semantics of FOOL). We will use program states as η-interpretations for FOOL terms. For example we will write $\mathrm{eval}_{st}(t)$ 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 $\mathrm{eval}_{st}(\varphi) = 1$.

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 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.*

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

□

Lemma 1. $\mathrm{eval}_{st}(\mathcal{T}(e)) = [\![\, e \,]\!]\,(st)$ *for each expression e and state st.* □

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

Proof. By structural induction on e. □

Definition 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 $\mathrm{updates}(s_1) \cup \mathrm{updates}(s_2) = \{x_1, \ldots, x_n\}$.
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 1. $\mathrm{eval}_{st}(\mathcal{N}(s)(t)) = \mathrm{eval}_{[\![\, s \,]\!](st)}(t)$ *for each statement s, state st and FOOL term t.* □

Proof. By structural induction on s. □

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 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 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 Experiments

In this section we describe our experiments on comparing the performance of the Vampire theorem prover [10] on FOOL and on translations of program properties to FOL. We used a collection of 50 programs written in the Boogie verification language [12]. 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 [2]. We generated these problems by running the front end of the Boogie [1] verifier.
2. 50 FOOL problems with tuples generated by running our implementation of the translation from Section 3.3, named Voogie.
3. 50 FOOL problems generated by running the BLT [3] 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 4.1) and discuss our set of benchmarks (Section 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 4.3. Our experimental results are summarised in Tables 1–3 and discussed in Section 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 under-specified state of the program.
2. Construct a guarded loop iteration i as $\texttt{if}\ \neg e\ \texttt{then}\ \mathit{bad} := \mathit{true}\ \texttt{else skip}\,; s$.
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.

4.2 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 [15] 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 [4] 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 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) [3] 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.

4.3 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 [17] 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 [13] 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.

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 [14] using the time limit of 5 minutes per problem. The detailed experimental results are available at `http://www.cse.chalmers.se/~evgenyk/ijcar18/`.

Table 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

Table 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	–	–	–

Tables 1 and 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 3 summarises the results of Vampire on the BLT translations of the benchmarks.

- 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.

Table 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	–	–	–

Based on the results presented in Tables 1–3 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.

5 Related Work

Our previous work introduced FOOL [8], its implementation in Vampire [6], and an efficient clausification algorithm for FOOL formulas [7].

In [6] 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 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 [12] and the automated verification framework [1]. The Boogie verifier encodes the next state relations of imperative programs in first-order logic by naming intermediate states of program variables [11].

BLT [3] 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 [6].

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.

Acknowledgments

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 EPSRC grants EP/K032674/1 (ReVeS) and EP/P03408X/1 (QuTie).

References

1. 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.
2. Clark Barrett, Aaron Stump, and Cesare 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`.
3. 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. 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. 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.
6. Evgenii Kotelnikov, Laura Kovács, Giles Reger, and Andrei Voronkov. The Vampire and the FOOL. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs, 2016*, pages 37–48, 2016.
7. 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.
8. Evgenii Kotelnikov, Laura Kovács, and Andrei Voronkov. A first class boolean sort in first-order theorem proving and TPTP. In *Intelligent Computer Mathematics*, pages 71–86. Springer, 2015.
9. 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.
10. Laura Kovács and Andrei Voronkov. First-Order Theorem Proving and Vampire. In *Proceedings of CAV*, volume 8044 of *LNCS*, pages 1–35, 2013.
11. K. Rustan M. Leino. Efficient weakest preconditions. *Inf. Process. Lett.*, 93(6):281–288, 2005.
12. K. Rustan M. Leino. This is Boogie 2. *Manuscript KRML*, 178(131), 2008.
13. 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.
14. Aaron Stump, Geoff Sutcliffe, and Cesare Tinelli. StarExec: A cross-community infrastructure for logic solving. In *Automated Reasoning – 7th International Joint Conference, IJCAR 2014*, pages 367–373, 2014.
15. Geoff Sutcliffe. The TPTP problem library and associated infrastructure — from CNF to TH0, TPTP v6.4.0. *J. Autom. Reasoning*, 59(4):483–502, 2017.
16. Geoff Sutcliffe, Stephan Schulz, Koen Claessen, and Peter Baumgartner. The TPTP Typed First-Order Form with Arithmetic. In *Proceedings of LPAR*, volume 7180 of *LNCS*, pages 406–419, 2012.
17. Andrei Voronkov. AVATAR: the architecture for first-order theorem provers. In *Computer Aided Verification — 26th International Conference, CAV 2014*, pages 696–710, 2014.