EPiC Series in Computing

Volume 41, 2016, Pages 1–19

GCAI 2016. 2nd Global Conference on Artificial Intelligence

A Clausal Normal Form Translation for FOOL*

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

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

Abstract

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

1 Introduction

Automated theorem provers for first-order logic usually operate on sets of first-order clauses. In order to check a formula in full first-order logic, theorem provers first translate it to clausal normal form (CNF). It is well-known that the quality of this translation affects the performance of the theorem prover. While there is no absolute criterion of what the best CNF for a formula is, theorem provers usually try to make the CNF smaller according to some measure. This measure can include the number of clauses, the number of literals, the lengths of the clauses and the size of the resulting signature, i.e. the number of function and predicate symbols. Implementors of CNF translations commonly employ formula simplification [12], (generalised) formula naming [12, 1], and other clausification techniques, aimed to make the CNF smaller.

Our recent work [9] presented a modification of many-sorted first-order logic with first-class boolean sort. We called this logic FOOL, standing for first-order logic (FOL) with boolean sort. FOOL extends standard FOL by (i) treating boolean terms as formulas, (ii) `if-then-else` expressions, (iii) `let-in` expressions, and (iv) tuple expressions. While `if-then-else` and

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

`let-in` expressions are also available in the SMT-LIB core language [3], the standard input language for SMT solvers, FOOL is a strict superset of SMT-LIB as tuple expressions are not part of SMT-LIB and `let-in` expressions in FOOL can define non-constant functions and predicate symbols.

There is a model-preserving translation of FOOL formulas to FOL (see [9]) that works by replacing parts of a FOOL formula with applications of fresh function and predicate symbols and extending the set of assumptions with definitions of these symbols. To reason about a FOOL formula, one can thus first translate it to a FOL formula and then convert the FOL formula into a set of clauses using the usual first-order clausification techniques. While this translation provides an easy way to support FOOL in existing first-order provers, it is not necessarily efficient. A more efficient translation can convert a FOOL formula directly to a set of first-order clauses, skipping the intermediate step of converting FOOL to FOL. This way, the translation can integrate known clausification techniques and improve the quality of the resulting clausal normal form.

In this paper we present a new clausification algorithm, called $\text{VCNF}_{\text{FOOL}}$, that translates a FOOL formula to an equisatisfiable set of first-order clauses. Our algorithm avoids producing large numbers of duplicate clauses and new symbols during clausification and also avoids clauses that can make theorem provers inefficient. We show that in practice this leads to a significant increase in the performance of a theorem prover).

Our $\text{VCNF}_{\text{FOOL}}$ algorithm is a non-trivial extension of the recent VCNF clausification algorithm for FOL [13]. The extension employs several clausification techniques for handling the non-FOL features of FOOL, namely boolean terms and `if-then-else`, `let-in` and tuple expressions. These techniques comprise the contributions of this work and are listed below.

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

1. We present a new clausification algorithm for translating FOOL formulas to an equisatisfiable set of first-order clauses.
2. We handle boolean variables in FOOL formulas by skolemising them using skolem predicates instead of skolem functions, thus avoiding the introduction of new boolean equalities.
3. We control the clausification of FOOL formulas with `if-then-else` and `let-in` expressions by a threshold level on the number of formula occurrences. Depending on the threshold, our algorithms decides on the fly whether to inline `if-then-else` and `let-in` expressions or introduce a new name and definition for them.
4. We handle tuple expressions in FOOL by introducing so-called projection functions and use these projection functions in the translation of `let-in` expressions with tuple definition.
5. We implemented our work in the Vampire theorem prover [10], offering this way an automated support to reason about FOOL formulas.
6. We evaluate our work on three benchmark suites coming from verification and analysis of software and described in Section 4, and show experimentally that our method significantly improves over [8] by the number of solved problems and the runtime.

2 Clausal Normal Form for First-Order Logic

Traditional approaches to clausification in FOL [12, 10] produce a clausal normal form in several stages, where each stage represents a single pass through the formula tree. These stages may include formula simplification, translation into (equivalence) negation normal form, formula naming, elimination of equivalences, skolemisation, and distribution of disjunctions over conjunctions. The VCNF clausification algorithm of [13] takes a different approach and employs a single top-down traversal of the formula in which these stages are combined. This enables optimisations that are not available if the stages of clausification are independent. For example, compared to the traditional staged approach, VCNF can introduce fewer skolem functions on formulas with complex nesting of equivalences and quantifiers. Moreover, it can detect and discard intermediate tautologies, which are much more difficult to recognise by the staged approach.

In this paper we use the VCNF algorithm and extended it to a new clausification algorithm for FOOL [9]. The main advantage of VCNF for our work, however, is that its top-down traversal provides a suitable context not only for clausification of first-order formulas, but also of the extension of first-order logic with FOOL features. In this section we overview the main features of VCNF. We will follow the notation used in [13] and in what follows will repeat some of the definitions.

2.1 Preliminaries

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

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

A *formula* is of the form $\varphi_1 \wedge \ldots \wedge \varphi_n$, $\varphi_1 \vee \ldots \vee \varphi_n$, $\varphi_1 \Rightarrow \varphi_2$, $\varphi_1 \leftrightarrow \varphi_2$, $\varphi_1 \otimes \varphi_2$, $\neg\varphi_1$, $\exists x : \tau.\varphi_1$, $\forall x : \tau.\varphi_1$, $\bot$, $\top$, or l where φ_i are formulas, x is a variable, τ a sort and l is a literal. Note that we treat conjunction and disjunction as n-ary operators; we assume that formulas are kept in *flattened form*, e.g. $(\varphi_1 \wedge \varphi_2) \wedge \varphi_3$ is always represented as $\varphi_1 \wedge \varphi_2 \wedge \varphi_3$. Furthermore, we assume that usage of $\top$ and $\bot$ is simplified immediately.

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

2.2 VCNF

The VCNF algorithm [13] works with finite sets of sequents. During computation the algorithm may construct substitutions to be applied to existing (signed) formulas. It is convenient for us to collect these substitutions without immediately applying them. For this reason, instead of a sequent $D\theta$, where θ is a substitution, we will use pairs D_θ consisting of a sequent D and a substitution θ. We will (slightly informally) also refer to such pairs as sequents.

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

The replacements of sequents are guided by the structure of φ. VCNF traverses φ top-down, processing every non-atomic subformula of φ exactly once in an order that respects the subformula relation. That is, for each two distinct subformulas ψ_1 and ψ_2 of φ such that ψ_1 is a subformula of ψ_2, ψ_2 is processed before ψ_1. For every subformula of φ, VCNF maintains a list of its occurrences as signed formulas in the sequents of C. The occurrences are updated whenever sequents are removed from and added to C. The main role of the list is to allow for a fast enumeration and lookup of all the occurrences when a particular subformula is to be processed. As explained below, the number of occurrences is also used to decided whether a subformula should be named. The replacements are governed by a set of rules that are, essentially, the standard tableau rules for first-order logic. We briefly summarise these rules below, and refer to [13] for details.

We note that except for the rule for negation, which essentially flips the sign of each occurrence of $\psi = \neg\gamma$ and replaces ψ with its immediate sub-formula γ in all the sequents, the remaining rules come in pairs in which they are dual to each other. For instance, dealing with a disjunction $\gamma_1 \vee \gamma_2$ with a positive $\star = \mathtt{t}$ is analogous to dealing with a conjunction with a negative sign. For simplicity, we only show the versions for $\star = \mathtt{t}$ below.

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

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

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

Whenever the number of occurrences of a subformula ψ in sequents in C exceeds a pre-specified *naming threshold*, ψ is named as follows. Let $y_1, \ldots, y_n$ be free variables of ψ and $\tau_1, \ldots, \tau_n$ be their sorts. VCNF introduces a new predicate symbol P of the sort $\tau_1 \times \ldots \times \tau_n$.

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

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

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

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

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

3 Clausal Normal Form for FOOL

In this section we describe our new clausification algorithm for FOOL. The algorithm takes a FOOL formula as input and produces an equisatisfiable set of first-order clauses. We write $\text{VCNF}_{\text{FOOL}}$ to refer to this algorithm, and FOOL2FOL to refer to the algorithm of [9] for translating FOOL formulas to arbitrary FOL formulas. In what follows, we first briefly overview the FOOL logic and then describe $\text{VCNF}_{\text{FOOL}}$ and compare the CNFs produces by it and FOOL2FOL.

3.1 FOOL

FOOL [9] extends the standard many-sorted FOL with an interpreted boolean sort. Boolean variables can be used as formulas in FOOL and formulas may be used as arguments to function and predicate symbols. In addition to its first-class boolean sort, FOOL extends standard FOL with following constructs:

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

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

1. $(s_1, \ldots, s_n)$, where $s_1, \ldots, s_n$ are terms;
2. `if` φ `then` s_1 `else` s_2, where s_1 and s_2 are tuple expressions;

3. a `let-in` expression of the form `let` D `in` t, where D is tuple, function, or predicate definition, and t is a tuple expression.

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

There are several ways to support the interpreted boolean sort in first-order theorem proving. The approach taken in [9] proposes to axiomatise it by adding two constants *true* and *false* of this sorts and two axioms: $true \not\doteq false$ and $(\forall x : bool)(x \doteq true \vee x \doteq false)$. Furthermore, [9] proposes a modification in superposition calculus of first-order provers: it (i) changes the simplification ordering of first-order prover by making *true* and *false* the smallest terms of boolean sort and (ii) replaces the second axiom with a so-called FOOL paramodulation rule. These modifications block self-paramodulation of $x \doteq true \vee x \doteq false$ and hence prevent performance problems arising from self-paramodulation in superposition theorem proving. In this paper, we however argue that neither boolean axiom nor modifications of superposition calculus are needed to support the interpreted boolean sort. Rather, we propose special processing of boolean variables and boolean equalities during clausification and avoid the introduction of new boolean equalities.

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

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

- We allow sequents to contain signed FOOL formulas, and not just first-order formulas.
- We extend the VCNF tautology elimination with the support for boolean variables. Whenever a boolean variable occurs in a sequent twice with the opposite signs, that sequent is not added to C. Whenever a boolean variable occurs in a sequent multiple times with the same sign, only one occurrence is kept in the sequent.
- We add extra rules that guide how sequents are replaced in the set C detailed below. These rules correspond to syntactical constructs available in FOOL but not in ordinary first-order logic.
- We change the rule that translates existentially quantified formulas to skolemise boolean variables using skolem predicates and not skolem functions. For that, we also allow substitutions to map boolean variables to skolem literals.
- We add an extra step of translation. After the input formula has been traversed, we apply substitutions of boolean variables to every formula in each respective sequent. The resulting set of sequents might have skolem literals occurring as terms. We run the clausification algorithm again on this set of sequents. The second run does not introduce new substitutions and results with a set of sequents that only contain atomic formulas and substitutions of non-boolean variables.

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

- We do not distinguish formulas used as arguments as a separate syntactical construct, but rather treat each such formula φ as an `if-then-else` expression of the form `if` φ `then` *true* `else` *false*.
- We assume that every `let-in` expression defines exactly one function or predicate symbol. Every `let-in` expression that defines more than one symbol can be transformed to multiple nested `let-in` expressions, each defining a single symbol, possibly by renaming some of the symbols.
- We assume that `let-in` expressions only occur as formulas. Every atomic formula that contains a `let-in` expression can be transformed to a `let-in` expression that defines the same symbol and occurs as a formula.
- Finally, we assume that each function or predicate symbol is defined by a `let-in` expression at most once. This can be achieved by a standard renaming policy.

3.3 $\mathrm{VCNF}_{\mathrm{FOOL}}$ Rules

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

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

Boolean Variables

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

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

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

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

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

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

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

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

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

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

Next, $\mathrm{VCNF}_{\mathrm{FOOL}}$ eliminates the tautology $\mathit{false}^{\mathtt{t}}$ in both sequents. The literal $P(\mathit{sk})$ contains a formula inside, therefore $\mathrm{VCNF}_{\mathrm{FOOL}}$ translates it as the formula $P(\texttt{if}\ \mathit{sk}\ \texttt{then}\ \mathit{true}\ \texttt{else}\ \mathit{false})$ according to the rules given in Section 3.3:

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

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

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

FOOL2FOL converts φ to the following set of clauses:

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

where sk is a skolem constant of the sort bool. ❏

The FOOL2FOL algorithm of [9] replaces each boolean variable x occurring as formula with $x \doteq \mathit{true}$ and skolemises boolean variables using boolean skolem functions. Unlike FOOL2FOL, $\mathrm{VCNF}_{\mathrm{FOOL}}$ skolemises boolean variables using skolem predicates and substitutes boolean variables that do not need skolemisation with constants true and false. The approach taken in $\mathrm{VCNF}_{\mathrm{FOOL}}$ is superior in two regards.

1. FOOL2FOL converts each skolemised boolean variable x occurring as formula to an equality between skolem terms and true. $\mathrm{VCNF}_{\mathrm{FOOL}}$ converts x to a skolem literal which can be handled by standard superposition more efficiently.
2. Substitution of a universally quantified boolean variable with true and false can decrease the size of the translation. If the boolean variable occurs as formula, after applying the substitution, the occurrence is either removed or the whole sequent is discarded by tautology elimination in $\mathrm{VCNF}_{\mathrm{FOOL}}$.

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

1. It can only contain boolean variables and constants true and false as boolean terms.

   Every boolean term that occurs in φ is translated as formula and no boolean terms other than variables, true and false are introduced.

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

2. It does not contain equalities between boolean terms.

   Every boolean equality occurring in the input is translated as equivalence between its arguments, and no new boolean equalities are eventually introduced.

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

Boolean Equalities

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

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

`if-then-else` Expressions as Terms

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

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

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

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

Example. Consider a definition of the *max* function using `if-then-else` taken from [8]:

$$(\forall x : \mathbb{Z})(\forall y : \mathbb{Z})(max(x, y) \doteq \texttt{if}\ x \geq y\ \texttt{then}\ x\ \texttt{else}\ y). \tag{1}$$

To translate (1), VCNF$_{\text{FOOL}}$ first applies twice the rule for existential quantifier inherited from VCNF, obtaining the sequent $\{(max(x, y) \doteq \texttt{if}\ x \geq y\ \texttt{then}\ x\ \texttt{else}\ y)^{\mathtt{t}}\}_\epsilon$. Then, either expanding or naming processes the result.

- Expanding results in $\{(x \geq y)^{\mathtt{f}}, (max(x, y) \doteq x)^{\mathtt{t}}\}_\epsilon, \{(x \geq y)^{\mathtt{t}}, (max(x, y) \doteq y)^{\mathtt{f}}\}_\epsilon$.
- Naming results in

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

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

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

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

FOOL2FOL translates (1) to $(\forall x : \mathbb{Z})(\forall y : \mathbb{Z})(max(x, y) \doteq g(x, y))$, where g is a fresh function symbol defined by the following formulas:

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

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

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

❏

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

To balance between these two strategies, we introduce a parameter to $\text{VCNF}_{\text{FOOL}}$ called the `if-then-else` expansion threshold. By default, we heuristically set the `if-then-else` expansion threshold of $\text{VCNF}_{\text{FOOL}}$ to $\log_2 n$, where n is the naming threshold of VCNF. The `if-then-else` expansion threshold of $\text{VCNF}_{\text{FOOL}}$ limits the maximal number of expanded `if-then-else` expressions inside one atomic formula. We start by expanding all `if-then-else` expression and once the expansion threshold is reached, $\text{VCNF}_{\text{FOOL}}$ names the remaining `if-then-else` expressions.

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

To compare to FOOL2FOL, we recall that FOOL2FOL replaces each non-boolean `if-then-else` expression with an application of a fresh function symbol and adds the definition of the symbol to the set of assumptions. The definition is expressed as an equality. Unlike FOOL2FOL, our new $\text{VCNF}_{\text{FOOL}}$ algorithm avoids introducing new equalities and uses predicate guards for naming, thus avoiding possible self-paramodulation triggered by equality literals.

`if-then-else` Expressions as Formulas

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

If done unconditionally, the translation of nested `if-then-else` expressions could lead to an exponential increase in the number of sequents, as the condition formula χ is being copied.

However, VCNF$_{\text{FOOL}}$ inherits from VCNF the mechanism for naming subformulas with many occurrences (as explained in the previous section) which prevents such blowup.

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

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

To process (2), VCNF$_{\text{FOOL}}$ first applies twice the rule for existential quantifier inherited from VCNF, obtaining the sequent $\{(\texttt{if}\ max(x, y) \doteq x\ \texttt{then}\ x \geq y\ \texttt{else}\ y \geq x)^{\mathsf{t}}\}_{\epsilon}$. Then, VCNF$_{\text{FOOL}}$ applies the rule for the `if-then-else` expression:

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

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

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

In contrast, FOOL2FOL introduces a name for the `if-then-else` expression and translates (2) to $(\forall x : \mathbb{Z})(\forall y : \mathbb{Z})P(x, y)$, where P is a fresh predicate symbol of the sort $\mathbb{Z} \times \mathbb{Z}$ with the following definitions:

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

These three formulas ultimately yield the following set of clauses:

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

❏

`let-in` Expressions

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

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

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

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

`let-in` Expressions with Tuple Definitions

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

1. introduces a fresh sort τ, a fresh function symbol t of the sort τ, a fresh function symbol g of the sort $\tau_1 \times \ldots \times \tau_n \to \tau$, and n fresh function symbols $\pi_1,\ldots,\pi_n$ (called projection functions), where for each $1 \leq i \leq n$, π_i is of the sort $\tau \to \tau_i$;
2. adds a sequent to C that is obtained from D by replacing every occurrence of $\psi^\star$ with $(\texttt{let}\ t\ =\ s'\ \texttt{in}\ \gamma')^\star$. γ' is obtained from γ by replacing each free occurrence of c_i with $\pi_i(t)$ for each $1 \leq i \leq n$. s' is obtained from s by replacing every tuple expression $(s_1,\ldots,s_n)$ with $g(s_1,\ldots,s_n)$;
3. adds sequents to C that axiomatise functions $g, \pi_1,\ldots,\pi_n$. In particular, these state that $\pi_i(g(s_1,\ldots,s_n)) \doteq s_i$ for every $i = 1,\ldots,n$ and that $t_1 \doteq t_2 \leftrightarrow \bigwedge_{i=1}^{n} \pi_i(t_1) \doteq \pi_i(t_2)$.

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

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

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

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

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

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

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

FOOL2FOL, as described in [9], cannot handle `let-in` expression with tuple definitions.

4 Experimental Results

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

In what follows, we report on our experimental results obtained by running Vampire on FOOL problems. Whenever we refer to Vampire, we mean the Vampire version extended with our new VCNF$_{\text{FOOL}}$ clausification algorithm for FOOL. We will write Vampire $\star$ for the

Table 1: Runtimes in seconds of provers on the set of 57 unsatisfiable algebraic datatypes problems.

Prover	Solved	Total time on solved problems
Vampire	56	23.470
Vampire $\star$	56	31.121
Z3	53	3.615
CVC4	53	25.480

previous version of Vampire with the FOOL2FOL algorithm of [8]; Vampire $\star$ translates FOOL formulas to FOL (after which they are clausified in a standard way) and uses a special inference rule to avoid FOOL self-paramodulation.

For our experiments, we used three sets of benchmarks: (i) problems taken from [14] on reasoning about (co)algebraic datatypes (see Sect. 4.1), (ii) examples with both quantifiers and uninterpreted functions taken from the SMT-LIB library [4] (see Sect. 4.2), and (iii) benchmarks on proving the partial correctness of loop-free programs (see Sect. 4.3). The last benchmark suite is constructed by us to illustrate the use of FOOL in program analysis and verification. As Vampire is the only automated first-order theorem prover supporting FOOL, and in particular `if-then-else` and `let-in` expressions, we could not compare Vampire with any other first-order prover. Further, Vampire $\star$ did not yet support tuple expressions in FOOL. Tuple expressions are also not supported by state-of-the-art SMT solvers. For these reasons, we compared Vampire against Vampire $\star$ and the SMT solvers CVC4 [2] and Z3 [6] only on the experiments from Sect. 4.1–4.2.

4.1 Experiments with Algebraic Datatypes Problems

We used 152 problems about (co)algebraic datatypes taken from [14]. These examples were generated by Isabelle and translated by us to the TPTP syntax [18]. These examples are expressed in FOOL, as they use boolean variables occurring as formulas, formulas occurring as arguments to function and predicate symbols, and `if-then-else` expressions. None of the 152 problems use `let-in` expressions.

We evaluated the performance of Vampire, Vampire $\star$, CVC4 and Z3 on the unsatisfiable problems in this set. In order to filter out satisfiable problems, we run all the provers on all the problems and only recorded the runs where at least one of the provers reported unsatisfiability. That gave us 57 problems.

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

Table 1 summarises our results. They were obtained on a MacBook Pro with a 2,9 GHz Intel Core i5 and 8 Gb RAM, with a 60 seconds time limit for each benchmark. Vampire and Vampire $\star$ solved the largest number of problems, both provers solved the same problems. 51 problems were solved by all provers. Both Vampire and Vampire $\star$ solved 3 problems, not solved by either CVC4 or Z3. CVC4 and Z3 solved one problem, not solved by either Vampire or Vampire $\star$. Compared to Vampire $\star$, Vampire showed significantly smaller runtime. We therefore conclude that our clausification algorithm for FOOL improved the performance of Vampire on this set of problems.

Table 2: Runtimes in seconds of provers on the set of 2191 unsatisfiable SMT-LIB problems.

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

4.2 Experiments with SMT-LIB Problems

As explained in more detail later on (see Section 5), FOOL can be regarded as a superset of the SMT-LIB core logic. A theorem prover that supports FOOL can be straightforwardly extended to read problems written in the SMT-LIB syntax. For our experiments using SMT-LIB problems, we used problems in quantified predicate logic with uninterpreted functions stored in the UF subspace of SMT-LIB. These problems use `if-then-else` expressions, `let-in` expressions that define constants, and formulas occurring as arguments to equality. None of the problems use quantifiers over the boolean sort. The problems taken from SMT-LIB are written in the SMT-LIB 2 syntax. In order to read these problems, we implemented a parser for a sufficient subset of the SMT-LIB 2 language in Vampire. The implementation of the parser comprised about 2,500 lines of C++ code.

We evaluated the performance of Vampire, Vampire $\star$, and CVC4 on unsatisfiable problems of the UF subspace. Each problem in the SMT-LIB library is marked with one of the statuses `sat`, `unsat` and `unknown`. A problem is marked as `sat` or `unsat` when at least two SMT solvers proved it to be satisfiable or unsatisfiable, respectively. Otherwise, a problem is marked as `unknown`. In order to filter out satisfiable problems, we ran Vampire, Vampire $\star$, and CVC4 on the problems marked as `unsat` and `unknown` and then recorded the results on the problems that were proven unsatisfiable by at least one prover. That gave us 2191 problems.

We ran Vampire twice on each problem: once with naming of `let-in` expressions and once with inlining (see Sect. 3.3). For each run the naming threshold was set to 8. In both runs we also used the option `--mode casc`. For each problem, we recorded the fastest successful run of Vampire. We ran Vampire $\star$ once on each problem with the option `--mode casc`.

Table 2 summarises the results of our experiments on the SMT-LIB problems. These results were obtained on the StarExec compute cluster [17] using the time limit of 5 minutes per problem. CVC4 solved the largest number of problems, Vampire solved significantly more than Vampire $\star$, and Z3 solved the least number of problems. None of the provers solved a superset of problems solved by another prover. The “Uniquely solved” column of Table 2 shows the number of problems that were solved by each of the provers, but not any of the other ones. 1675 problems were solved by all of the provers, and 2190 problems were solved by at least one of the provers. Vampire solved 111 problems not solved by Vampire $\star$, and Vampire $\star$ solved 19 problems not solved by Vampire.

We also recorded how different translations of `let-in` affected the performance of Vampire. Vampire with inlining of `let-in` expressions solved 61 problems not solved by Vampire without inlining of `let-in` expressions. Vampire without inlining of `let-in` expressions solved 45 problems not solved by Vampire without inlining of `let-in` expressions.

Based on the results of this experiment we make the following observations. Vampire solved new problems by inlining `let-in` expressions and expanding `if-then-else` expressions. Vampire could not solve some of the problems that were solved by Vampire $\star$, we explain it by the fact

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

Figure 1: The count_two program.

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

Figure 2: The benchmark obtained by unrolling Figure 1 three times.

that Vampire$\star$ always names `if-then-else` expressions, which turns out to be important for solving some problems. Both inlining and naming of `let-in` expressions can make a prover inefficient.

4.3 Experiments with FOOL Reasoning about Programs

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

We used five small programs with loops annotated with a safety property using the `assert` command. They are listed in Appendix A. Each program contains one loop with one or more `if-then-else` expressions, assignments and tests over integers, integer arrays and booleans. Table 3 summarises the programs used in our experiments: the programs `count_two`, `count_two_flag` and `count_three` implement versions of counting elements in an input array using different criteria and ensure that the sum of counted elements equal to the array length; `two_arrays` and `three_arrays` sort and compare two, and respectively three arrays element-wise. We unrolled these program loops 2, 3, 4 and 5 times, resulting in a set of 20 annotated loop-free programs. Figure 2 shows the program obtained by unrolling three times the loop of `count_two`.

For each one of the 20 loop-free benchmarks, we expressed its partial correctness as a TPTP problem using FOOL in the combination of the theory of linear integer arithmetic and the polymorphic theory of arrays [8]. To this end, (i) we formulated the safety assertion as a TPTP conjecture and (ii) expressed the transition relation of the program as a FOOL formula with tuple expressions and `let-in` expressions with tuple definitions. We refer to [8] for the details of the translation of a program's transition relation to FOOL. In particular, the correctness of this translation is stated in Theorem 1 of that work. Each FOOL formula produces by the translation is linear in the size of the program. Figure 3 shows the TPTP translation of the safety property of the `cout\_two\_tptp` program. It uses the `thf` subset of the TPTP language, which is the standard subset that contains features of FOOL.

The results of the experiments are summarised in Table 3. These results were obtained on a MacBook Pro with a 2,9 GHz Intel Core i5 and 8 Gb RAM, and using the time limit of 60 seconds per problem. The first column of the table lists the names of the programs with loops, and columns 2–5 indicate how many time the program loop was unrolled and gives the time needed by Vampire to prove the correctness of the corresponding loop-free program.

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

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

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

Figure 3: A FOOL translation of Fig. 2 written in the TPTP language.

Table 3: Runtimes in seconds of Vampire on 20 problems encoding partial program correctness.

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

5 Related Work

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

Most notably, Wisniewski et al. propose in [20] methods for normalising formulas in higher-order logic (HOL). Similarly to FOOL, HOL natively contains the boolean sort. Wisniewski et al. deal with formulas occurring at argument positions by a technique called *argument extraction* which, similarly to our naming schemes, extends the signature and defines a new symbol outside the original formula. Moreover, also Wisniewski et al. introduce skolem predicates instead of skolem functions when dealing with existential boolean quantifiers. This happens implicitly for them, since in HOL there is no fundamental distinction between formulas and terms.

FOOL can be regarded as a superset of SMT-LIB [3] core logic and formulas of SMT-LIB core logic can be directly expressed in FOOL. The language of FOOL extends the SMT-LIB core language with local function definitions, using `let-in` expressions defining functions of arbitrary, and not just zero, arity.

Despite the similarity of the languages, the technology used by modern SMT solvers [11] differs greatly from that of first-order theorem provers and so do the approaches to normalising the input formula. In particular, as SMT solvers pass the propositional abstraction of the input formula to an efficient SAT solver there is no great need to optimise extensions of the signature and clausification usually follows the simple Tseitin encoding [19] of the formula tree. Moreover, modern SMT solvers employ an alternative approach to dealing with quantifiers over interpreted sorts such as the booleans, which is complementary to skolemisation and relies on a guidance by counter-examples [15] or on model-based projections [5].

Finally, it is interesting to note that our $\text{VCNF}_{\text{FOOL}}$ algorithm naturally translates a quantified boolean formula (QBF), as realised in the FOOL language, into a CNF in effectively propositional logic (EPR). Specifically, every literal in this translation is a skolem predicate applied to boolean variables and constants *true* and *false*. This result is very close to the one proposed in [16], where the authors explicitly focus on QBF as the source and EPR as the target language, respectively. Obtaining a formula in EPR is a desirable property since there are first-order proving methods known to be efficient for dealing with the fragment (see e.g. [7]).

6 Conclusion and Future Work

Applications of program analysis and verification rely on SAT/SMT solvers and/or theorem provers to reason about program properties formulated in various logics. The efficiency of SAT/SMT solvers and theorem provers critically depends on the used clausification algorithm. In this paper we presented a new clausification algorithm, called $\text{VCNF}_{\text{FOOL}}$, for formulas expressed in FOOL. Our algorithm is a non-trivial extension of the recent VCNF clausification algorithm for standard first-order logic. $\text{VCNF}_{\text{FOOL}}$ for FOOL introduces skolem predicates over boolean variables, avoids equalities over boolean variables, and uses formula naming and tautology elimination on complex formulas. It also avoids excessively duplicating clauses and introducing too many new symbols. Thanks to the our new $\text{VCNF}_{\text{FOOL}}$ algorithm, proving FOOL formulas requires neither an axiomatisation of the boolean sort nor modifications in superposition calculus. We implemented our work in Vampire and experimentally showed its benefits on a large number of examples. For future work we are interested in developing further criteria for controlling naming and inlining expressions during clausification. Using FOOL for more complex applications of program analysis is another interesting venue to exploit.

Acknowledgments

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

References

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

[2] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Proceedings of CAV*, pages 171–177, 2011.

[3] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[4] 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`.

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

[6] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Proceedings of TACAS*, volume 4963 of *LNCS*, pages 337–340, 2008.

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

[8] 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.

[9] 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.

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

[12] Andreas Nonnengart and Christoph Weidenbach. Computing small clause normal forms. *Handbook of Automated Reasoning*, 1:335–367, 2001.

[13] Giles Reger, Martin Suda, and Andrei Voronkov. New techniques in clausal form generation. In Christoph Benzmüller, Raul Rojas, and Geoff Sutcliffe, editors, *GCAI 2016. Global Conference on Artificial Intelligence*, EasyChair Proceedings in Computing. EasyChair, 2016.

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

[15] Andrew Reynolds, Tim King, and Viktor Kuncak. An instantiation-based approach for solving quantified linear arithmetic. *CoRR*, abs/1510.02642, 2015.

[16] Martina Seidl, Florian Lonsing, and Armin Biere. qbf2epr: A tool for generating EPR formulas from QBF. In Pascal Fontaine, Renate A. Schmidt, and Stephan Schulz, editors, *Third Workshop on Practical Aspects of Automated Reasoning, PAAR-2012, Manchester, UK, June 30 - July 1, 2012*, volume 21 of *EPiC Series in Computing*, pages 139–148. EasyChair, 2012.

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

[18] Geoff Sutcliffe. The TPTP Problem Library and Associated Infrastructure. *Journal of Automated Reasoning*, 43(4):337–362, 2009.

[19] G.S. Tseitin. On the complexity of derivation in propositional calculus. In Jörg H. Siekmann and Graham Wrightson, editors, *Automation of Reasoning*, Symbolic Computation, pages 466–483. Springer Berlin Heidelberg, 1983.

[20] Max Wisniewski, Alexander Steen, Kim Kern, and Christoph Benzmüller. Effective normalization techniques for HOL. In Nicola Olivetti and Ashish Tiwari, editors, *Automated Reasoning - 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27 - July 2, 2016, Proceedings*, volume 9706 of *Lecture Notes in Computer Science*, pages 362–370. Springer, 2016.

A Imperative Programs with Loops and `if-then-else`

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

count_two

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

count_two_flag

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

count_three

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

two_arrays

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

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

three_arrays