Loop Analysis by Quantification over Iterations

Loop Analysis by Quantification over Iterations

Bernhard Gleiss[1], Laura Kovács[1,2], and Simon Robillard[2]

[1] TU Wien, Austria
[2] Chalmers University of Technology, Sweden

Abstract

We present a framework to analyze and verify programs containing loops by using a first-order language of so-called extended expressions. This language can express both functional and temporal properties of loops. We prove soundness and completeness of our framework and use our approach to automate the tasks of partial correctness verification, termination analysis and invariant generation. For doing so, we express the loop semantics as a set of first-order properties over extended expressions and use theorem provers and/or SMT solvers to reason about these properties. Our approach supports full first-order reasoning, including proving program properties with alternation of quantifiers. Our work is implemented in the tool QuIt and successfully evaluated on benchmarks coming from software verification.

1 Introduction

One of the major challenges in automating the analysis and verification of programs comes with the presence of loops. Reasoning about such programs requires inferring and proving non-trivial properties that describe the loop behavior. Loop properties can be categorized into two classes: (i) functional properties that describe the loop behavior on program states and summarize, e.g., partial correctness properties of the loop, and (ii) temporal properties that focus on the iterative behavior of the loop, in particular its termination. To analyze loops and reason about their behavior, it is often useful to consider properties that blur the distinction between those two categories, such as safety and liveness properties. While there has been tremendous work on analyzing and verifying program loops, see e.g., [11, 16, 18, 19, 24, 26, 29], traditional means to reason about imperative programs are still poorly equipped to deal with both types of properties in a uniform manner. Complex functional properties are commonly expressed as program assertions featuring quantifiers. For example, the program `reverse` given in Figure 1 copies the elements of an array a to an array b, reversing their order. To specify this behavior, we need to use a universally quantified property, e.g., the post-condition:

$$\forall j\, (0 \leq j < a.size \Rightarrow b[j] \approx a[a.size - 1 - j])$$

In some cases, we even need to use properties with quantifier alternations to give precise program specifications. The program `find-max-up-to` (also given in Figure 1) computes, for every position of an array a, the maximum value stored in a up to that position, and stores that value in b. One the properties that describe its specification is:

$$\forall j \exists k\, (0 \leq j < a.size \Rightarrow b[j] \approx a[k])$$

These quantified properties can be verified using, e.g., Hoare logic. On the other hand, temporal properties are best expressed in some form of temporal logic, which usually restricts the use of quantifiers. Furthermore, most verification techniques require program annotations such as invariants and termination measures to be provided by the programmer, which limits their potential for automation.

```
i := 0 ;
while i < a.size do
    b[i] := [a.size − 1 − i];
    i := i + 1 ;
end
```

(a) `reverse`

```
i := 0;
m := 0;
while i < a.size do
    if a[i] ⩾ a[m] then
        m := i;
    end
    b[i] := a[m];
    i := i + 1;
end
```

(b) `find-max-up-to`

Figure 1: Motivating examples over arrays with first-order properties.

In this paper, we present a framework to verify properties that combine functional and temporal aspects. The method is based on the *first-order language of extended expressions*, which provides a rich way to express both temporal and functional loop properties, including full quantification over loop iterations and program values. The semantics of a given loop can be encoded as a formula in this language, thus providing an axiomatization for the set of properties that hold for this loop (Section 3). Extended expressions are more expressive than program assertions typically used in program analysis and verification: program assertions reason about single program states, whereas extended expression correspond to properties over sequences of states, i.e., program traces.

By expressing the loop semantics as a set of extended expressions, we reduce various applications of program analysis and verification to problems of first-order logic. In particular, we show that partial correctness and termination properties of loops can naturally be expressed as extended expressions. Similarly, the problem of invariant generation is a special instance of first-order reasoning about extended expressions. Namely, by using consequence finding and symbol elimination over extended expressions in first-order theorem proving we automatically infer first-order loop invariants (Section 4).

Analyzing loops in our framework is thus reduced to the problem of reasoning about extended expressions. This problem can be solved by automated reasoning engines, such as first-order theorem provers and SMT solvers. We describe how encoding the loop semantics into extended expressions can be optimized for these tools, in particular by limiting the need to perform inductive reasoning and exploiting reasoning with both theories and quantifiers (Section 5).

To illustrate the practical application of our framework, we implemented our work in the tool QuIt, which translates programs into extended expressions and uses automated reasoning engines to prove these properties (Section 6). We evaluate our work on verification problems taken from related works [6,14] as well as from the array manipulating program category of the software verification benchmark suite SV-Comp [5]. For that, we used QuIt in conjunction with the first-order prover Vampire [25] and the SMT solvers CVC4 [3] and Z3 [13]. We show that, unlike existing methods, our approach supports reasoning about first-order loop properties with arbitrary use of quantifiers. We support quantification over loop iterations and program values, generating and proving loop properties in full first-order theories. By using our framework of extended expression, we are able to prove the safety assertions of each example of Figure 1.

Contributions. Extended expressions were first introduced for invariant generation in [24] and later used in [1] for proving partial correctness of programs. The work presented in this

paper extends this line of work and brings the following contributions:

1. We formalize the semantics of the language of extended expressions;
2. We describe the axiomatization of the theory of extended expressions that hold for a given loop and prove its completeness (up to completeness of the background theory);
3. We show how extended expressions can be used to express and verify functional and temporal properties about programs, in particular partial correctness and termination;
4. We prove the soundness of using symbol elimination for invariant generation, based on extended expressions and consequence finding in first-order theorem proving;
5. We experiment with different background theories, in particular arrays and natural numbers, and compare different provers on these encodings.

2 Preliminaries

2.1 First-order logic

We consider standard many-sorted first-order logic modulo a background theory. We denote the theory with T and its signature with Σ_T. We assume that Σ_T includes sorts, equality $\approx$ over each sort and the interpreted functions and predicates of linear arithmetic $0, 1, +, <$. For example, in one set of our experiments, T is the combined theory of linear integer arithmetic and arrays. We assume a given domain, and a mapping of the symbols in Σ_T to this domain. Any interpretation that respects that mapping is called a T-interpretation. Semantic consequence under T is denoted $A \models_T B$, i.e., any T-interpretation that satisfies A satisfies B.

We call a closed first-order formula a sentence. The valuation of sentences is defined in the usual way. In particular for the valuation of quantified sentences, we consider extensions of interpretations to variables: given an interpretation $\mathcal{I}$, we denote by $\mathcal{I}[x \leftarrow d]$ the interpretation that extends $\mathcal{I}$ by mapping the variable x to the domain value d. The sentence $\forall x\ \phi$ (resp. $\exists x\ \phi$) is true in $\mathcal{I}$ if ϕ is true in $\mathcal{I}[x \leftarrow d]$ for any (resp. some) value d of the appropriate domain.

2.2 Program semantics

Throughout this paper, we assume a given loop $L = (C, \pi)$, where π is a program corresponding to the loop body and C is a Boolean expression representing the loop condition. The finite set of program locations[1] occurring in π and C is denoted by **Loc**.

We do not consider a particular programming language for π. Instead, we only rely on the denotational semantics of π, defined as a transition relation on program states. A state is a mapping from program locations to values of the appropriate sort. The semantics of π is described by the relation $\mathcal{S}_\pi$: for any pair of states (σ, σ'), the pair belongs to $\mathcal{S}_\pi$ if the execution of π in state σ can lead to state σ'. If π is a deterministic program, $\mathcal{S}_\pi$ is a function, but in general we do not assume this property. We require $\mathcal{S}_\pi$ to be total, but this does not limit our framework to loops with terminating bodies. If the loop body π is not guaranteed to terminate, we can use a special state $\sigma_\perp$ to represent non-terminating computations, with the requirement that $(\sigma_\perp, \sigma) \in \mathcal{S}_\pi$ if and only if $\sigma = \sigma_\perp$.

[1] We do not use the term "program variables", in order to avoid confusion with variables occurring in formulas.

Definition 1 (L-sequence)**.** An L-sequence is an infinite sequence of states $\sigma_0, \sigma_1, \ldots$ such that for any natural number i, $(\sigma_i, \sigma_{i+1}) \in \mathcal{S}_\pi$.

The set of L-sequences correspond to all possible executions of the loop L. A non-terminating loop execution corresponds to an L-sequence in which the condition C is true in all states σ_i. Terminating loop executions correspond to a prefix $\sigma_0, \ldots, \sigma_k$ of an L-sequence such that the condition C is true in the states σ_0 to σ_{k-1} and false in σ_k. The requirement on $\mathcal{S}_\pi$ to be total is necessary so that L-sequences only include infinite sequences (which is needed to provide a full interpretation of the language of extended expressions, see Section 3.1).

In practice, for most languages, $\mathcal{S}_\pi$ can easily be computed when π does not itself contain loops. In the presence of nested loops, it is possible to rely on an over-approximation of the actual semantics relation. If invariants are given for the nested loop, they can be used to improve the accuracy of the approximation. This approach is sound in the sense that, for every possible execution of the loop, there exists an L-sequence. However, in the rest of this paper, we assume that $\mathcal{S}_\pi$ is exact, and for concrete examples, we consider only non-nested loops.

2.3 Language of assertions

The central idea of our work is the use of a language of *extended expressions* in which formulas can express properties of executions of the loop L, i.e., L-sequences. Extended expressions are more expressive than the kind of assertions traditionally used in program verification: program assertions express properties of a single program state, whereas *extended expressions describe sequences of states (that is, traces)*. For establishing a correspondence between extended expressions and assertions, we formally define a language for assertions, denoted by $\mathcal{L}_{asrt}$.

The signature of the language $\mathcal{L}_{asrt}$ is $\Sigma_T \cup \Sigma_{asrt}$, where Σ_{asrt} is the set that includes a constant symbol $\mu_l : \tau$ for every location l in **Loc**, where τ is the sort corresponding to the type of the location l.

Definition 2. Given a program state σ , the σ-interpretation is the unique interpretation $\mathcal{I}$ for $\mathcal{L}_{asrt}$ such that:

1. $\mathcal{I}$ is a T-interpretation;

2. $\mathcal{I}(\mu_l) = \sigma(l)$ for each program location $l \in$ **Loc**.

If a sentence F is true in the σ-interpretation, we write $\vDash_\sigma F$. Using Hoare triple notation, we write $\{P\}\,\pi\,\{Q\}$ to denote that, for any state σ, if $\vDash_\sigma P$, then for any state σ' such that $(\sigma, \sigma') \in \mathcal{S}_\pi$, $\vDash_{\sigma'} Q$.

Definition 3. $\mathcal{L}_{asrt}$ is said to be *expressive with respect to* π if for every formula F in $\mathcal{L}_{asrt}$, there exists a sentence $\mathrm{pre}_\pi(F)$ with the following property: for states σ, if F is true in the σ-interpretation (possibly extended to some variables), then, for all states σ', $\mathrm{pre}_\pi(F)$ is true in the (similarly extended) σ'-interpretation if and only if $(\sigma', \sigma) \in \mathcal{S}_\pi$.

Remark 1. The above definition indicates that $\mathcal{L}_{asrt}$ can be used to express the weakest pre-condition of π. It is not possible to prove this property without limiting π to specific programming constructs and specifying the theory T, but in practice many assertion languages are expressive. We note that, as in [30], we could define the expressivity of $\mathcal{L}_{asrt}$ by requiring the existence of a strongest post-condition.

3 Extended expressions

We now define the *first-order language of extended expressions*. The semantics of a program can be expressed in this language and used further as an axiomatization for the set of valid program properties.

3.1 Syntax and semantics

The language of extended expressions is denoted $\mathcal{L}_{extd}$. Its signature is $\Sigma_T \cup \Sigma_{extd}$, where Σ_{extd} includes, for every location l in **Loc**, a function symbol $\nu_l : \mathbb{N} \to \tau$, where τ is a sort corresponding to the type of the location l. We call these symbols *extended symbols* and use the notation $\nu_l^{(i)}$ to denote the application of an extended symbol ν_l to a term i. The semantics of $\mathcal{L}_{extd}$ is based on the possible executions of the loop L.

Definition 4. Given an infinite sequence of states $\bar{\sigma} = \sigma_0, \sigma_1, \ldots$ (that is not required to have the properties of an L-sequence), the *$\bar{\sigma}$-interpretation* is the unique interpretation $\mathcal{I}$ such that:

1. $\mathcal{I}$ is a T-interpretation;
2. $\mathcal{I}(\nu_l) = f_l$ for each location $l \in$ **Loc**, where f_l is a function such that for any $i \in \mathbb{N}$, $f_l(i) = \sigma_i(l)$.

If, for a sequence $\bar{\sigma}$, a sentence F in $\mathcal{L}_{extd}$ is true under the $\bar{\sigma}$-interpretation, we write $\vDash_{\bar{\sigma}} F$. If for all L-sequences $\bar{\sigma}$, we have $\vDash_{\bar{\sigma}} F$, then we say that F is *L-valid*, denoted $\vDash_L F$. Intuitively, L-valid sentences are the properties that are true for all executions of L.

3.2 Relativised formulas

We now describe how to obtain a formula in $\mathcal{L}_{extd}$ corresponding to an assertion in $\mathcal{L}_{asrt}$.

Definition 5 (Relativised formula)**.** Given a (possibly open) formula F in $\mathcal{L}_{asrt}$ and a term t of sort $\mathbb{N}$, we define the *relativised formula*, denoted $F^{(t)}$, as the formula obtained by replacing every occurrence of a symbol $\mu_l \in \mathcal{L}_{asrt}$ from F by the term $\nu_l^{(t)}$.

For example given a term i of sort $\mathbb{N}$ and a formula $F = \exists x, \mu_l \approx 2 \times x$, the relativised formula $F^{(i)}$ is $\exists x, \nu_l^{(i)} \approx 2 \times x$. The relativised formula is in $\mathcal{L}_{extd}$, and the set of variables occurring free in it is exactly the set of variables occurring free in F or in t.

Lemma 1 (Semantics of relativised formula)**.** *Let F be a formula in $\mathcal{L}_{asrt}$, $\bar{\sigma}$ an infinite sequence of states, and m a natural number. Let $\mathcal{I}$ denote the $\bar{\sigma}$-interpretation. The value of $F^{(t)}$ under $\mathcal{I}[t \leftarrow m]$ is identical (for any interpretation of the free variables) to the value of F under the σ_m-interpretation.*

Proof. By induction on the syntactic structure of F. For the base case, it is easy to check that any term in F has the same interpretation as the corresponding term in the relativised formula. □

3.3 Axiomatization of Valid Loop Properties

Let us consider the theory of L-valid sentences, i.e. the set of sentences $F \in \mathcal{L}_{extd}$ such that $\vDash_L F$. In order to axiomatize this theory, we need to encode in $\mathcal{L}_{extd}$ the semantics of π, and

thus describe L-sequences. Provided that $\mathcal{L}_{asrt}$ is expressive with respect to π, the semantics of the loop (ignoring its condition) can be described by the following axiom:

$$\forall \bar{x_l}, i \ \left(S^{(i+1)} \Rightarrow \mathrm{pre}_\pi(S)^{(i)} \right) \qquad (\mathsf{Step}_L)$$

where S is the formula $\bigwedge_l \mu_l \approx x_l$, and $\bar{x_l}$ is a set of distinct variables (one for each location $l \in \mathbf{Loc}$).

For example, let us consider the following loop:

```
while a ≠ b do
|  if a > b then
|  |  a := a − b;
|  else
|  |  b := b − a;
|  end
end
```

The set of locations read or modified by the loop is $\{a, b\}$, therefore the formula S is $x \approx \mu_a \wedge y \approx \mu_b$. Using a typical predicate transformer calculus, we can compute the weakest pre-condition $\mathrm{pre}_\pi(S) = (\mu_a > \mu_b \Rightarrow x \approx \mu_a - \mu_b \wedge y \approx \mu_b) \wedge (\neg \mu_a > \mu_b \Rightarrow x \approx \mu_a \wedge y \approx \mu_b - \mu_a)$. Therefore the axiom Step_L for this particular loop is

$$\forall x, y, i \ \Big(x \approx \nu_a^{(i+1)} \wedge y \approx \nu_b^{(i+1)} \Rightarrow \Big(\nu_a^{(i)} > \nu_b^{(i)} \Rightarrow x \approx \nu_a^{(i)} - \nu_b^{(i)} \wedge y \approx \nu_b^{(i)} \Big) \\ \wedge \Big(\neg \nu_a^{(i)} > \nu_b^{(i)} \Rightarrow x \approx \nu_a^{(i)} \wedge y \approx \nu_b^{(i)} - \nu_a^{(i)} \Big) \Big)$$

We see that Step_L can equivalently be expressed without using variables for locations, in this case:

$$\forall i \ \Big(\nu_a^{(i)} > \nu_b^{(i)} \Rightarrow \nu_a^{(i+1)} \approx \nu_a^{(i)} - \nu_b^{(i)} \wedge \nu_b^{(i+1)} \approx \nu_b^{(i)} \Big) \\ \wedge \Big(\neg \nu_a^{(i)} > \nu_b^{(i)} \Rightarrow \nu_a^{(i+1)} \approx \nu_a^{(i)} \wedge \nu_b^{(i+1)} \approx \nu_b^{(i)} - \nu_a^{(i)} \Big)$$

This simplification is desirable in practice as it limits the number of quantifiers. We will however consider the syntactic form presented above in order to keep the presentation simple.

Lemma 2 (Soundness)**.** $\vDash_L \mathsf{Step}_L$.

Proof. Let $\bar{\sigma} = \sigma_0, \sigma_1, \ldots$ be an L-sequence and $\mathcal{I}$ the $\bar{\sigma}$-interpretation, we show that Step_L is true in $\mathcal{I}$.

Let m be a natural number and let $\bar{d}$ be values of the domain corresponding to variables $\bar{x_l}$. Let $\mathcal{I}'$ be the interpretation $\mathcal{I}[i \leftarrow m+1, \bar{x_l} \leftarrow \bar{d}]$. If $S^{(i+1)}$ is false in $\mathcal{I}'$, the formula $S^{(i+1)} \Rightarrow \mathrm{pre}_\pi(S)^{(i)}$ is true in $\mathcal{I}'$. Otherwise, by Lemma 1, it must be the case that S is true in the σ_{m+1}-interpretation (extended with the interpretation $\bar{d}$ of the free variables $\bar{x_l}$). Since $(\sigma_m, \sigma_{m+1}) \in \mathcal{S}_\pi$, we have that $\vDash_{\sigma_m} \mathrm{pre}_\pi(S)$, therefore $\mathrm{pre}_\pi(S)^{(i)}$ is true in $\mathcal{I}'$. The formula Step_L is true in $\mathcal{I}$ for any interpretation of its quantified variables. □

Lemma 3 (Completeness)**.** *Let F be a sentence in $\mathcal{L}_{extd}$ such that $\vDash_L F$, then $\mathsf{Step}_L \vDash_T F$.*

Proof. Let $\mathcal{I}$ be a T-interpretation that satisfies Step_L. We define $\bar{\sigma} = \sigma_0, \sigma_1, \ldots$ to be the infinite sequence of states such that for any number i and any program location $l \in \mathbf{Loc}$, $\sigma_i(l) = f_l(i)$, where f_l is $\mathcal{I}(l)$. Clearly $\mathcal{I}$ is the $\bar{\sigma}$-interpretation. Let us show that $\bar{\sigma}$ is a L-sequence.

Let m be a number, and let $\bar{d}$ denote the values of all the program locations $l \in \mathbf{Loc}$ in state σ_{m+1}. Let $\mathcal{I}'$ be the interpretation $\mathcal{I}[i \leftarrow m, \bar{x}_l \leftarrow \bar{d}]$. By choice of $\bar{d}$, $\vDash_{m+1} S$ with the σ_{m+1}-interpretation extended with $[\bar{x}_l \leftarrow \bar{d}]$. Thus by Lemma 1, $S^{(i+1)}$ is true in $\mathcal{I}'$. Since $\mathcal{I}'$ satisfies Step_L, $\mathrm{pre}_\pi(S)^{(i)}$ is in particular true in $\mathcal{I}'$. By Lemma 1, we have $\vDash_{\sigma_m} \mathrm{pre}_\pi(S)$ (with the same extension of the σ_m-interpretation as before). By definition of $\mathrm{pre}_\pi(S)$, this implies $(\sigma_m, \sigma_{m+1}) \in \mathcal{S}_\pi$, therefore $\bar{\sigma}$ satisfies the definition of an L-sequence.

Since $\mathcal{I}$ is a $\bar{\sigma}$-interpretation derived from an L-sequence, it must satisfy F. □

Theorem 4 (L-validity)**.** *For any sentence F in $\mathcal{L}_{extd}$, $\vDash_L F$ if and only if $\mathsf{Step}_L \vDash_T F$.*

Proof. One direction of the equivalence is given by Lemma 3. For the other direction, let $\bar{\sigma}$ be an L-sequence and $\mathcal{I}$ the $\bar{\sigma}$-interpretation. By Lemma 2, $\mathcal{I}$ is a model of Step_L. In addition, $\mathcal{I}$ is a T-interpretation, therefore by the assumption it is also a model of F. □

Remark 2. The theory of L-valid sentences is a superset of the theory T. Therefore in order for that theory to be complete, T must be complete as well. Theorem 4 shows that this is indeed a sufficient condition. In that sense, it can be seen as a result of relative completeness.

4 Applications of Extended Expressions

We now detail how applications of program analysis and verification can be expressed as problems over extended expressions. In particular, we show that proving partial correctness or termination of programs can be reduced to proving properties of extended expressions. Further, the task of invariant generation can be solved by using consequence finding and symbol elimination in first-order theorem proving over extended expressions.

4.1 Verifying partial loop correctness

The partial correctness of the loop L with respect to a pre-condition P and a post-condition Q (both sentences in $\mathcal{L}_{asrt}$) and the loop condition C can be expressed as a sentence in $\mathcal{L}_{extd}$:

$$\forall n \left(P^{(0)} \wedge \forall m \left(m < n \Rightarrow C^{(m)} \right) \wedge \neg C^{(n)} \right) \Rightarrow Q^{(n)} \qquad (\mathsf{Correct})$$

Lemma 5 (Partial correctness of the loop)**.** *If $\vDash_L \mathsf{Correct}$, then for any state σ satisfying P, any terminating execution of the loop L starting in σ leads to a state that satisfies Q.*

Proof. Let $\sigma_0, \ldots, \sigma_k$ be a finite sequence of states corresponding to a terminating execution of L, such that $\sigma_0 = \sigma$. For any two consecutive states σ_i and σ_{i+1} in the sequence, $(\sigma_i, \sigma_{i+1}) \in \mathcal{S}_\pi$. Let $\bar{\sigma}$ be an L-sequence such that $\sigma_0, \ldots, \sigma_k$ is a prefix of $\bar{\sigma}$, and $\mathcal{I}$ the $\bar{\sigma}$-interpretation. By the assumption, $\mathcal{I}$ is a model of $\mathsf{Correct}$. Let $\mathcal{I}'$ be the interpretation $\mathcal{I}[n \leftarrow k]$.

By $\vDash_{\sigma_0} P$ and Lemma 1, we have that $P^{(0)}$ is true in $\mathcal{I}'$. By property of the finite execution of the loop $\sigma_0, \ldots, \sigma_k$, the loop condition C is true in every state of the sequence except the last. Therefore, $\vDash_{\sigma_k} \neg C$ and $\vDash_{\sigma_i} C$ for any number i such that $i < k$. Using Lemma 1 we can check that $\forall m \left(m < n \Rightarrow C^{(m)} \right)$ and $\neg C^{(n)}$ are true in $\mathcal{I}'$. Consequently $Q^{(n)}$ is true in $\mathcal{I}'$, and by Lemma 1, $\vDash_{\sigma_k} Q$, that is, the final state of the execution satisfies Q. □

4.2 Termination, safety, liveness

Similarly, proving the termination of L under a pre-condition P can be reduced to checking the L-validity of the sentence

$$P^{(0)} \Rightarrow \exists n,\ \neg C^{(n)} \tag{Termin}$$

Lemma 6 (Termination of the loop)**.** *If* $\vDash_L$ Termin*, then for any state* σ *satisfying* P*, any execution of the loop* L *starting in* σ *terminates.*

Proof. By contradiction let $\bar{\sigma}$ be an L-sequence corresponding to a non-terminating execution of L, i.e., all its states σ_i verify $\vDash_{\sigma_i} C$, and such that $\sigma_0 = \sigma$. Let $\mathcal{I}$ be the $\bar{\sigma}$-interpretation. Since $\vDash_{\sigma_0} P$, by Lemma 1, $P^{(0)}$ is true in $\mathcal{I}$, and since $\mathcal{I}$ is a model of Termin, it is in particular a model of $\exists n\ \neg C^{(n)}$. By Lemma 1 there exists a number k such that $\vDash_{\sigma_k} \neg C$, which contradicts our hypothesis. □

Finally, it is possible to express safety and liveness properties as extended expressions. Given an assertion A, the safety property with respect to A is

$$\forall n,\ \neg A^{(n)} \tag{Safe}$$

and the liveness property

$$\forall m \exists n\ \left(m < n \wedge A^{(n)}\right) \tag{Live}$$

It is easy to check that these formulas correspond to the expected semantic properties of the loop, in a fashion similar to the proofs of lemmas 5 and 6.

4.3 Invariant generation via symbol elimination

Our framework provides a way to verify the correctness of an iterative program without the explicit use of invariants. Nevertheless, it can be useful to obtain loop invariants for the program, e.g., to gain some insight in the behavior of the loop or to verify it using another tool, which typically requires invariants in the form of user annotations. Our framework can be used to derive invariants as logical consequences of the extended expressions.

In program verification, the notion of invariant typically refers to *inductive* invariants, i.e. assertions F such that $\{C \wedge F\}\,\pi\,\{F\}$. In practice, interesting invariants are those that hold at the start of the loop. Therefore in the presence of a pre-condition P, we wish to find invariants F such that $P \vDash_T F$. Given these two requirements, we use the following definition:

Definition 6 (P-invariant)**.** Given a sentence P in $\mathcal{L}_{asrt}$, a sentence F in $\mathcal{L}_{asrt}$ is a P-invariant if for any prefix $\sigma_0, \ldots, \sigma_k$ of an L-sequence such that $\vDash_{\sigma_0} P$ and $\vDash_{\sigma_i} C$ for any number $i < k$, then any state σ of that prefix verifies $\vDash_\sigma F$.

Intuitively, if an execution of the loop L starts in a state satisfying P, every state of this execution satisfies F, up to and including the final state if the loop terminates. It is easy to show that an inductive invariant I is a P-invariant for any sentence P such that $P \vDash_T I$. Conversely however, not all P-invariants F are inductive: there may exist a pair of states (σ, σ') that violates $\{C \wedge F\}\,\pi\,\{F\}$, but only if σ is not reachable in any L-sequence that starts with a state satisfying P.

Lemma 7. *Let* P *and* F *be sentences in* $\mathcal{L}_{asrt}$*. Let* Inv *denote the sentence*

$$\forall n\ \left(P^{(0)} \wedge \forall m\ \left(m < n \Rightarrow C^{(m)}\right) \Rightarrow F^{(n)}\right) \tag{Inv}$$

If $\vDash_L$ Inv*, then* F *is a* P*-invariant.*

Proof. Let $\bar{\sigma}$ be an L-sequence such that $\vDash_{\sigma_0} P$ and k be a number such that $\vDash_{\sigma_i} C$, for any number $i < k$. Let $\mathcal{I}$ be the $\bar{\sigma}$ interpretation and j be a number such that $j \leqslant k$. Since $\mathcal{I}[n \leftarrow j]$ is a model of Inv, it is in particular a model of $F^{(n)}$. Therefore, by Lemma 1, $\vDash_{\sigma_j} F$. □

Lemma 7 provides a way to check that a given sentence is a P-invariant. More interestingly, it also gives the basis of a procedure to generate P-invariants. This procedure was first introduced in [24] but never proven sound until now. We outline it here again in order to provide a formal argument for its soundness based on Lemma 7.

We have so far only considered reasoning about extended expressions. In order to generate P-invariants, we must now turn our attention to assertions, i.e., formulas in $\mathcal{L}_{asrt}$. Firstly, let us observe that for any formula F in $\mathcal{L}_{asrt}$ and any term t of sort $\mathbb{N}$, the relativised formula $F^{(t)}$ is equivalent to $\bigwedge_l \nu_l^{(t)} \approx \mu_l \Rightarrow F$ (regardless of the interpretation of the symbols μ_l). Thus Inv is equivalent to

$$\forall n \left(P^{(0)} \wedge \forall m \left(m < n \Rightarrow C^{(m)} \right) \wedge \bigwedge_{l \in \mathbf{Loc}} \nu_l^{(n)} \approx \mu_l \Rightarrow F \right)$$

Secondly, if F is a closed formula (in particular, not featuring any occurrence of the variable n), Inv can be rewritten so that the quantifier is moved to the antecedent

$$\exists n \left(P^{(0)} \wedge \forall m \left(m < n \Rightarrow C^{(m)} \right) \wedge \bigwedge_{l \in \mathbf{Loc}} \nu_l^{(n)} \approx \mu_l \right) \Rightarrow F$$

Let us denote by InvGen the sentence

$$\exists n \left(P^{(0)} \wedge \forall m \left(m < n \Rightarrow C^{(m)} \right) \wedge \bigwedge_{l \in \mathbf{Loc}} \nu_l^{(n)} \approx \mu_l \right) \qquad (\mathsf{InvGen})$$

By Theorem 4, the condition in Lemma 7 is equivalent to $\mathsf{Step}_L \cup \mathsf{InvGen} \vDash_T F$. In order to generate P-invariants, it is therefore enough to find consequences (under theory T) of $\mathsf{Step}_L \cup \mathsf{InvGen}$. Theorem provers based on saturation provide a natural way to perform consequence finding. The sentences Step_L and InvGen are clausified (in the process, a constant symbol n corresponding to the existentially quantified variable is introduced by Skolemization) and the resulting set of clauses is saturated: new clauses are repeatedly produced by (sound) inferences between clauses of the set, and the conclusion added to the set. Typically this process is used to derive the empty clause as part of a proof by contradiction. However in this case the initial set of clauses has a model, therefore the process will only stop if no new non-redundant clauses can be derived.

Any clause generated during saturation by a sound calculus is a logical consequence of the original set of clauses. To be a P-invariant, it must only contain symbols from $\mathcal{L}_{asrt}$. Unguided consequence finding is unlikely to yield such formulas consistently, so we take advantage of the reduction ordering of the superposition calculus to orient the search. Superposition is a family of calculii parametrized by a reduction ordering on terms. This ordering is used to restrict the number of possible inferences while preserving the refutational completeness of the calculus. For example the superposition rule

$$\frac{l \approx r \vee A \qquad B[l'] \vee C}{(A \vee B[r] \vee C)\theta}$$

which uses an equality literal $l \approx r$ to rewrite a term l' (unifiable with l under a most general unifier θ), is applied only if $l \not\succcurlyeq r$ according to the term ordering. By choosing the right term ordering, we can ensure that symbol-eliminating inferences are favored by the prover. In particular, terms featuring extended symbols should be larger than others. One possibility to accomplish this is to choose a Knuth-Bendix term ordering in which extended symbols are given a large weight and a large precedence.

The set of clauses resulting from the clausification of InvGen contains, for each program location l, the unit clause $\nu_l^{(n)} = \mu_l$, therefore any clause featuring a term unifiable with $\nu_l^{(n)}$ may be used in a symbol eliminating inference. Every time a new clause is generated that does not feature an extended symbol, it may be reported as an invariant. Optionally, it may be preferable to report such clauses only if they feature symbols from Σ_{asrt}. Clauses that do not include such symbols are theory tautologies: they are technically invariant but not very useful for program verification.

5 Automated Reasoning with Extended Expressions

As shown in Section 4, analyzing loops in our framework is reduced to the problem of reasoning about extended expressions, which can be solved by automated reasoning engines, such as first-order theorem provers and SMT solvers. In this section we describe how encodings of the loop semantics into extended expressions can be optimized for these tools.

5.1 Avoiding induction

Theorem 4 provides a powerful way to reduce loop analysis and verification tasks to the problem of proving that a certain extended expression is entailed by Step_L. Unsurprisingly, induction often plays a key role in these proofs. Step_L essentially describes the semantics of one arbitrary iteration of the loop, whereas loop analysis is often concerned with proving properties of arbitrary iterations, for example for all iterations or for a symbolic iteration in which the loop condition is negated for the first time.

Extending automated theorem provers with inductive reasoning is a very challenging problem, with some preliminary yet still limited results in [12, 21, 34]. In order to avoid inductive reasoning and thus make our framework more friendly to automated provers, we use a number of so-called *trace lemmas* in addition to Step_L. These lemmas correspond to valid properties of the loop.

Definition 7. A *trace lemma* for a given loop is a sentence F such that $\mathsf{Step}_L \models_T F$.

For any set of trace lemmas Lem_L, it is obvious that $\mathsf{Step}_L \cup \mathsf{Lem}_L$ is T-equivalent to Step_L. Hence, Step_L can be replaced by $\mathsf{Step}_L \cup \mathsf{Lem}_L$ in loop analysis, in particular in the applications described in Section 4. While this substitution makes no difference on a theoretical level, a careful choice of Lem_L often leads to a dramatic improvement of the performance of automated reasoning tools. The choice of trace lemmas to include in Lem_L depends on the theory T, the class of programs targeted, and the type of properties that one wishes to analyze. Consider for example the extended expression

$$\forall i, j \ \left(i < j \Rightarrow \nu_l^{(i)} \leqslant \nu_l^{(j)} \right)$$

Many loops include so-called monotonically increasing locations l for which this property would hold. In addition, the property is likely to be useful in many verification tasks. On the other

hand, proving that it is entailed by Step_L requires reasoning by induction and hence automated theorem provers are unlikely to discover it. For these reasons, the property is a good candidate trace lemma: every time we perform a verification task, we will first check that the property holds for each location in the given loop, and if so, add it to the set of trace lemmas that will later be used in conjunction with Step_L.

Proving that a given sentence F is a trace lemma of L is in general as difficult as proving the L-validity of other extended expressions. Therefore, we rely on sound but incomplete methods to derive trace lemmas of L. Currently in our work, the verification of trace lemmas is accomplished by lightweight static analysis techniques. For example, the property described above is added to the set of trace lemmas when all assignments to the location l are increments by a positive constant. A more general method to check whether a sentence is a trace lemma is to reduce it to a minimal condition on one iteration of the loop body π. The property given in example holds if and only if $\{\mu_l \approx x\}\,\pi\,\{\mu_l \geqslant x\}$ for some variable x. Simple properties such as these can often be verified automatically. This hints at a generic way to describe trace lemmas and use them: (i) an extended expression F (often a property universally quantified over iterations) is first proven to be equivalent to an easily verifiable condition on the loop body in the form of a Hoare triple (ii) any time the condition is verified, F is included in Lem_L.

A complete description of the set of trace lemmas used in our work is given in an earlier publication [22]. In general, our trace lemmas fall under two categories: properties of monotonically increasing or decreasing locations and properties of array updates.

5.2 Encoding of natural numbers

The theory T must include a sort of natural numbers, as well as the predicates and functions of linear arithmetic that are needed to formulate the axiom Step_L. However, not all automated tools for reasoning in first-order logic support the theory of natural numbers. Therefore we experimented with two encodings of natural numbers. Our first encoding uses integers, where every axiom or goal is modified to ensure that only non-negative integers are considered. The second encoding is based on a term algebra generated by two constructors, a constant *zero* and a unary function *succ*. The predicate $<$ is recursively axiomatized.

Both encodings will be handled differently by provers, and yield different proofs. Linear arithmetic is a staple of reasoning modulo theory, and all SMT solvers include a solver for it. For first-order theorem provers based on saturation, reasoning about linear arithmetic is traditionally accomplished by including a (partial) axiomatization in the set of clauses to saturate. Recently, these provers have taken advantage of SMT solvers to perform theory reasoning on ground clauses [31] as well as non-ground clauses [32]. Regarding term algebras, the SMT solvers Z3 and CVC4 both include theory solvers for the ground theory. The automated theorem prover Vampire allows reasoning with term algebras, based on a conservative extension of the theory complemented by dedicated inference rules [7, 23].

In addition, the two encodings allow the expression of different properties. The integer-based encoding can more easily express relations between integer-valued program locations and iteration numbers. For example, if a location l of the loop satisfies $\{\mu_l \approx x\}\,\pi\,\{\mu_l \approx x + c\}$ for some variable x and some constant c then the following trace lemma can be used:

$$\forall i \ \left(0 \leqslant i \Rightarrow \nu_l^{(i)} \approx \nu_l^{(0)} + i \times c\right)$$

If natural numbers are encoded as algebraic terms, an equivalent trace lemma cannot be expressed without extending the theory to include a function mapping natural numbers to

integers. Instead we can use a weaker property, for example (if $c \neq 0$):

$$\forall i, j \ \left(\nu_l^{(i)} \approx \nu_l^{(j)} \Rightarrow i \approx j\right)$$

5.3 Representation of arrays

Unlike scalar program locations, the logical encoding of arrays is not straightforward. In earlier approaches [24], we used the following functional representation of arrays. In $\mathcal{L}_{asrt}$, arrays are represented as functions from the sort of indices to the sort of values. In $\mathcal{L}_{extd}$, arrays are represented by binary functions: the first argument of the function takes an iteration and the second an index, so that $a(i, p)$ denotes the value stored at position p at the ith iteration.

Later experiments suggested that using a dedicated theory of arrays might make it easier to prove properties of programs [9]. In this setting, we have one sort τ for each type of array, equipped with operations store and select that represent writing to and reading from a array, respectively. Array locations are then treated like other locations: in $\mathcal{L}_{asrt}$ they are represented by constants of type τ, and in $\mathcal{L}_{extd}$ they are represented by a function from $\mathbb{N}$ to τ.

6 Experiments

6.1 Implementation

We implemented our work in the tool QuIt[2]. QuIt consists of 12000 lines of C++ code. Inputs to QuIt are programs written in a guarded command language. In addition to the program itself, the input also includes pre- and post-conditions (P and Q) in the form of first-order logic assertions with unbounded quantification. QuIt converts this program to a first-order problem, according to one of its three modes of operation:

- *Verification mode*, to prove partial program correctness (Section 4.1). In this setting, the first-order problem produced by QuIt contains the hypothesis Step_L, the trace lemmas and theory axioms, and the goal Correct to be proven.
- *Termination mode*, to prove program termination (Section 4.2). In this case, QuIt generates a similar problem as in its verification mode, but with the goal Termin.
- *Invariant generation mode*, to generate invariants by symbol elimination (Section 4.3). The problem produced contains the hypothesis InvGen, trace lemmas and theory axioms. Extended symbols are marked for symbol elimination. No goal to be proven is provided, since the aim is to produce consequences of properties of extended expressions, rather than to find a proof.

QuIt is partially based on code from [22] that was previously integrated in the first-order theorem prover Vampire. We made QuIt a standalone tool that can interact with various provers, including both SMT solvers and first-order theorem provers. For that, QuIt outputs problems in the TPTP syntax of first-order theorem provers [33], in particular in the TFF input representation of many-sorted first-order logic. Further, QuIt also generates its output in the SMT-LIB input syntax of SMT solvers [4]. As such, problems generated by QuIt in the verification and termination modes can be passed to any tool that supports the TPTP and/or SMT-LIB syntax. Currently, only Vampire is able to perform symbol elimination, which is necessary to handle the invariant generation problems generated by QuIt.

[2] http://www.cse.chalmers.se/~simrob/downloads/quit.tar.gz

Benchmark	Vampire				CVC4				Z3			
	A+T	A+I	F+T	F+I	A+T	A+I	F+T	F+I	A+T	A+I	F+T	F+I
absolute-prop1	✓	✓	✓	✓	t	t	t	t	t	t	t	t
absolute-prop2	✓	✓	t	✓	t	t	t	t	t	t	t	t
atleast-one-iteration	✓	t	✓	t	t	t	t	t	✓	✓	✓	✓
both-or-none	✓	✓	✓	✓	t	t	t	t	t	t	t	t
check-equal-set-flag	t	t	t	t	t	t	t	t	t	t	t	t
copy	✓	✓	✓	✓	t	t	t	t	t	t	t	t
copy-nonzero-prop1	t	t	t	t	t	t	t	t	t	t	t	t
copy-nonzero-prop2	t	t	t	t	t	t	t	t	t	t	t	t
copy-odd	✓	✓	✓	✓	t	t	t	t	t	t	t	t
copy-partial	✓	✓	✓	✓	t	t	t	t	t	t	t	t
copy-positive	t	t	t	t	t	t	t	t	t	t	t	t
copy-two-indices	t	✓	t	✓	t	t	t	t	t	t	t	t
find1-prop1	✓	t	✓	t	t	t	t	t	✓	✓	✓	✓
find1-prop2	✓	t	✓	t	t	t	t	t	t	t	t	t
find1-prop3	✓	✓	✓	✓	t	t	t	t	t	t	t	t
find2-prop1	✓	✓	✓	✓	✓	t	✓	t	✓	✓	✓	✓
find2-prop2	✓	✓	✓	✓	t	t	t	t	t	✓	t	t
find2-prop3	✓	✓	✓	✓	t	t	t	t	t	✓	t	t
find-max	t	t	t	t	t	t	t	t	t	t	t	t
find-max-up-to-prop1	t	t	t	t	t	t	t	t	t	t	t	t
find-max-up-to-prop2	✓	✓	✓	✓	t	t	t	t	t	t	t	t
find-max-from-second	t	t	t	t	t	t	t	t	t	t	t	t
find-min	t	t	t	t	t	t	t	t	t	t	t	t
find-min-up-to	✓	✓	✓	✓	t	t	t	t	t	t	t	t
find-sentinel	✓	✓	✓	✓	t	t	t	t	t	✓	t	t
find-two-max-prop1	t	t	t	t	t	t	t	t	t	t	t	t
find-two-max-prop2	t	t	t	t	t	t	t	t	t	t	t	✓
in-place-max	t	t	t	t	t	t	t	t	t	t	t	t
increment-by-one-prop1	✓	✓	✓	✓	t	t	t	t	t	t	t	t
increment-by-one-prop2	✓	✓	t	✓	t	t	t	t	t	t	t	t
indexn-is-arraylength	✓	✓	✓	✓	t	t	t	t	✓	✓	✓	✓
init	✓	✓	✓	✓	t	t	t	t	t	t	t	t
init-conditionally-prop1	t	t	t	t	t	t	t	t	t	t	t	t
init-conditionally-prop2	t	t	t	t	t	t	t	t	t	t	t	t
init-even	t	✓	t	✓	t	t	t	t	t	✓	t	t
init-non-constant	✓	✓	✓	✓	t	t	t	t	t	t	t	t
init-partial	✓	✓	✓	✓	t	t	t	t	t	✓	t	t
init-previous-plus-one	t	t	t	t	t	t	t	t	t	t	t	✓
max-prop1	✓	✓	✓	✓	t	t	t	t	t	t	t	t
max-prop2	✓	✓	✓	✓	t	t	t	t	t	t	t	t
merge-interleave-prop1	t	✓	t	✓	t	t	t	t	t	t	t	t
merge-interleave-prop2	t	t	t	t	t	t	t	t	t	t	t	t
palindrome	t	t	t	t	t	t	t	t	t	t	t	t
partition	t	t	t	t	t	t	t	t	t	t	t	t
partition-init	t	t	t	t	t	t	t	t	t	t	t	t
push-back-prop1	t	✓	t	✓	t	t	t	t	t	✓	t	✓
push-back-prop2	t	✓	t	✓	t	t	t	t	t	t	t	t
reverse	✓	✓	✓	✓	t	t	t	t	t	t	t	t
set-to-one	✓	t	✓	t	t	t	t	t	✓	✓	✓	✓
str-cpy	✓	✓	✓	✓	t	t	t	t	t	t	t	t
str-len	✓	✓	✓	✓	t	t	t	t	t	t	t	t
swap-prop1	t	t	t	t	t	t	t	t	t	t	t	t
swap-prop2	t	t	t	t	t	t	t	t	t	t	t	t
vector-addition	✓	✓	✓	✓	t	t	t	t	t	t	t	t
vector-subtraction	✓	✓	✓	✓	t	t	t	t	t	t	t	t
Total	35				1				13			
Unique	24				0				2			

Table 1: Results of theorems provers on QuIt-generated partial correctness problems. Success is denoted by ✓ and timeout by t.

6.2 Experimental results

To evaluate our implementation, we collected benchmarks from the work of [14] and the SV-Comp repository of software verification benchmarks [5]. We converted these examples manually into our input-format. Since our approach establishes program correctness rather than searching for counterexamples, we omitted benchmarks where assertions are violated. We also omitted examples not supported by our framework due to language features such as multiple loops or memory management. Lastly, we removed duplicate problems differing from other examples only in the names of program locations. As a result, our benchmarks include 55 test cases, all featuring arrays.

The assertion language used in these benchmarks does not allow quantification and relies on loops to encode some quantified properties. For example, the program fragment:

$$\texttt{for(int i = 0; i < a.length; ++ i) \{assert(F(a[i]))\}}$$

is used to encode the universal property $\forall i.(0 \leq i < a.length \implies F(a[i]))$. Using program code to encode first-order properties is however restrictive, as only universally quantified properties over finite domains can be naturally encoded. Since our framework supports unbounded quantification in first-order properties, we used quantified assertions rather than loops to describe the properties to verify.

To overcome the challenges of first-order reasoning with theories, quantifiers and induction (see Section 5), in QUIT we used four different encodings of the first-order background theory: (i) theory of arrays and term algebras (denoted by `A+T`), (ii) theory of arrays and linear integer arithmetic (denoted by `A+I`), (iii) first-order theory of term algebras with uninterpreted functions modeling arrays (denoted by `F+T`), and (iv) first-order theory linear of integer arithmetic with uninterpreted functions modeling arrays (denoted by `F+I`). That is, natural numbers were encoded either by term algebra axioms or by a sound, but incomplete axiomatization of linear integer arithmetic. By applying these four encodings to our 55 examples, QUIT produced all together 220 examples for each of its modes. To prove these examples, we interfaced QUIT with three solvers, namely VAMPIRE, CVC4 and Z3. We report on our experiments, which were performed on an Intel Core i5 machine running at 2.9Ghz.

Proving partial correctness. For each choice of background theory encoding, we used QUIT in the verification mode to construct a first-order formalization of partial correctness (which took less than a second for any benchmark) and then ran each of the provers on the resulting problem with a timeout of 60 seconds. Our results are summarized in Table 1. The first column of this table names the benchmark name as in SV-Comp. For each solver, we then report on its result on the problem generated by QUIT using one encoding of the background theory: ✓ denotes success (the prover proved the QUIT problem), while t denotes failure due to time-out. Table 1 also reports on the total number of problems solved by each prover, as well as on the number of problems that were uniquely solved by only one prover.

Table 1 shows that VAMPIRE outperforms both SMT solvers on problems created by QUIT, regardless of the options chosen. This likely stems from the use of many quantified properties among the trace lemmas. Concerning the background theory and the choice of encoding, it is difficult to identify a winning encoding. Each configuration was able to uniquely solve some problems. This suggests that a portfolio approach might be advisable: the different possible encodings of the problem can all be generated, and proof attempts may be conducted by different provers, possibly in parallel. In order to test the usefulness of trace lemmas, we also ran the partial correctness experiment without including any such lemmas, and instead only including the hypothesis Step_L. Only 3 programs could be proven correct in this setting, demonstrating the crucial role of trace lemmas.

Proving termination. We used QuIt in the termination mode to construct a first-order encoding of program termination (which again took less than a second for any benchmark). We ran each prover on the resulting problem with a timeout of 60 seconds. Some of the 55 examples differ only by their post-condition, which is irrelevant for termination, so our termination benchmarks include 43 different programs. Vampire was able to prove termination of 42 of these programs. The example for which termination could not be proven is `find1`, in which the loop condition depends on the value of a location set in the loop body. Z3 managed to prove termination of 23 programs, whereas CVC4 did not solve any of the termination problems. While our benchmarks do not yield challenging termination problems, they correspond to common programming patterns. We believe that the ability to check their termination automatically, in the same framework used to verify correctness, is of great practical use.

Generating invariants. To generate invariants, we interfaced QuIt only with Vampire since it is currently the only solver able to perform symbol elimination over first-order properties. Since the problem created in invariant generation mode is satisfiable, saturation may never terminate, generating an infinite set of logical consequences. Therefore we ran Vampire with a fixed time limit of 10 seconds on each QuIt problem. Our experiments show that Vampire was able to find invariants for all the problems. Depending on the background theory encoding used in the QuIt problem, Vampire generated between 411 and as many as 11000 clauses within the time limit, each clause representing a loop invariant. The criteria used to evaluate the quality of these invariants depends largely on the application. To verify partial correctness with respect to pre- and post-conditions P and Q, a common task is to use invariants to prove that Q holds after the execution of the loop (typically one would also need to prove that the invariant is true under P; this is guaranteed by our definition of P-invariants). In order to test the quality of our generated invariants for this application, we used the following procedure: we constructed a new first-order problem containing the generated invariants and the negation of the loop condition as hypotheses, and added the post-condition as the goal to be proven. We then ran Vampire on the resulting problems, by using it in portfolio mode with a time limit of 60 seconds.

For those benchmarks where partial correctness can be proven from the invariants, we can analyze the resulting proof in order to gather some interesting invariants. For example for program `reverse` (Figure 1) the following invariant was generated and later used to prove correctness:

$$\forall x,\ (x < 0 \ \vee \ \neg x < i \ \vee \ b[(a.length - 1) - x] \approx a[x])\,.$$

We were able to prove partial correctness from the generated invariants for 21 programs in total, a subset of the programs for which we were able to establish correctness using only extended expressions. The list of these programs is given in Table 2. Programs for which the post-condition could be proven by Vampire from the invariants generated are denoted by ✓.

7 Related work

Our definition of the theory of L-valid sentences is reminiscent of modal logics: we consider sentences that are true across a certain class of interpretations (one interpretation of each possible execution of L), akin to the multiple worlds used by Kripke semantics. However in our setting there is no notion of accessibility between those different worlds. This allows the use of first-order quantification, without the difficulties that are inherent in defining semantics for *first-order* modal logic [15]. In addition, automated reasoning for modal logics remains a difficult problem, despite efforts in that direction [28].

Benchmark	Vampire			
	A+T	A+I	F+T	F+I
absolute-prop1	t	t	t	t
absolute-prop2	t	t	t	t
atleast-one-iteration	t	t	t	t
both-or-none	t	t	t	t
check-equal-set-flag	t	t	t	t
copy	✓	✓	✓	✓
copy-nonzero-prop1	t	t	t	t
copy-nonzero-prop2	t	t	t	t
copy-odd	✓	✓	✓	✓
copy-partial	✓	✓	✓	✓
copy-positive	t	t	t	t
copy-two-indices	t	t	t	✓
find1-prop1	t	t	t	t
find1-prop2	t	t	t	t
find1-prop3	✓	✓	✓	✓
find2-prop1	✓	✓	✓	✓
find2-prop2	✓	✓	✓	✓
find2-prop3	✓	✓	✓	✓
find-max	t	t	t	t
find-max-from-second	t	t	t	t
find-max-up-to-prop1	t	t	t	t
find-max-up-to-prop2	t	t	t	t
find-min	t	t	t	t
find-min-up-to	t	t	t	t
find-sentinel	t	t	t	t
find-two-max-prop1	t	t	t	t
find-two-max-prop2	t	t	t	t
in-place-max	t	t	t	t
increment-by-one-prop1	✓	✓	✓	✓
increment-by-one-prop2	t	✓	t	✓
indexn-is-arraylength	t	✓	t	✓
init	✓	✓	✓	✓
init-conditionally-prop1	t	t	t	t
init-conditionally-prop2	t	t	t	t
init-even	t	t	t	t
init-non-constant	✓	✓	t	✓
init-partial	✓	✓	✓	✓
init-previous-plus-one	t	t	t	t
max-prop1	t	t	t	t
max-prop2	t	t	t	t
merge-interleave-prop1	t	✓	t	✓
merge-interleave-prop2	t	t	t	t
palindrome	t	t	t	t
partition	t	t	t	t
partition-init	t	t	t	t
push-back-prop1	t	✓	t	t
push-back-prop2	t	✓	t	✓
reverse	t	t	t	✓
set-to-one	t	t	t	t
str-cpy	✓	✓	✓	✓
str-len	✓	✓	✓	✓
swap-prop1	t	t	t	t
swap-prop2	t	t	t	t
vector-addition	t	✓	t	✓
vector-subtraction	✓	✓	✓	✓
Total	14	20	13	21
Unique	0	0	0	2

Table 2: Results of Vampire on proving partial correctness using invariants generated by symbol elimination.

Analyzing loops and generating quantified invariants has been addressed by a large number of approaches. One line of research iteratively generates quantifier-free properties that are generalized into universally quantified invariants. The work of [19] generates universally quantified inductive invariants by iteratively inferring and strengthening candidate invariants. The method uses SMT solving and is therefore restricted to first-order theories with a finite model property. SMT-based invariant generation is also performed in [18] and universal invariants with a bounded number of universal quantifiers are inferred. In [2], Craig interpolation over bounded loop executions is used to generate candidate ground invariants and terms to be universally quantified in those invariants. Candidate invariants are also used in the formula slicing approach of [20]. In [6], templates of quantified invariants are used to reduce the problem of quantified invariant generation to computing quantifier-free invariants. Template invariants together with SMT-based constraint solving is also used in [27] to generate universal invariants. Unlike these works, we are not limited to universal invariants but can infer first-order loop properties with alternations of quantifiers. First-order resolution has previously been used to derive invariants with alternations of quantifiers in [8]. In this work, the derivation is goal-oriented, whereas our technique does not require a post-condition to be given. The work of [29] relies on Craig interpolation in superposition theorem proving to generate quantified invariants. The approach is however restricted to universal invariants.

Our use of trace lemmas to guide the automation shares some similarity with template-based approaches for invariant generation [10, 17]. Our work, however, does not require any assumptions on the syntactic shape of the target invariants. Instead, assumptions are made about semantic patterns that are often shared across many programs. The invariants are not restricted to the shape of the trace lemmas, and the lemmas are discovered automatically, without user guidance. Moreover, our approach can be used with arbitrary first-order theories, even with theories that have no interpolation property and/or a finite axiomatization.

Another line of work focuses on the design of specialized abstract domains to represent and infer universal properties by abstract interpretation. The fluid updates abstraction of [14] creates pair-wise points-to relations over arrays and solves these constrains using SMT solving. The array segmentation domain of [11] reasons about the contents of an array by dividing it into consecutive subsets of array elements. These methods are very expressive, but limited to their respective abstract domains, and to universal invariants. For example, the abstraction domain used in [11] would not be able to handle `reverse` program from Figure 1. Rather than performing a custom, domain-specific analysis, our work introduces a generic first-order framework for deriving and proving first-order loop properties.

8 Conclusion

We described a logical framework for expressing and proving complex properties of loops. Our framework is based on the first-order language of extended expressions and supports full first-order quantification over both program values and iterations. We showed how to use our work to automate various tasks of program analysis and verification, in particular by using our approach in conjunction with automated reasoning techniques in first-order logic. For future work, we plan to extend our programming model by considering various background theories. For example, the theory of term algebras could be used to reason about programs with recursive data structures. Another interesting question is whether our semantics of iterations can be extended to support nested and consecutive loops in a way that remains tractable for automated theorem provers.

Acknowledgements. We thank Wolfgang Ahrendt, Martin Suda and Andrei Voronkov for fruitful discussions leading up to this work, as well as the anonymous referees for their suggestions. This work was funded by the ERC Starting Grant 2014 SYMCAR 639270, the Wallenberg Academy Fellowship 2014 TheProSE, the Swedish Research Council grant GenPro D0497701, and the Austrian FWF research project RiSE S11409-N23.

References

[1] Wolfgang Ahrendt, Laura Kovács, and Simon Robillard. Reasoning about loops using vampire in KeY. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 434–443. Springer, 2015.

[2] Francesco Alberti, Roberto Bruttomesso, Silvio Ghilardi, Silvio Ranise, and Natasha Sharygina. SAFARI: SMT-based abstraction for arrays with interpolants. In *International Conference on Computer Aided Verification*, pages 679–685. Springer, 2012.

[3] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *International Conference on Computer Aided Verification*, pages 171–177. Springer, 2011.

[4] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at `www.SMT-LIB.org`.

[5] Dirk Beyer. Software verification with validation of results. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 331–349. Springer, 2017.

[6] Nikolaj Bjørner, Ken McMillan, and Andrey Rybalchenko. On solving universally quantified Horn clauses. In *International Static Analysis Symposium*, pages 105–125. Springer, 2013.

[7] Jasmin Christian Blanchette, Nicolas Peltier, and Simon Robillard. Superposition with datatypes and codatatypes. In *Automated Reasoning*, pages 370–387. Springer, 2018.

[8] Ritu Chadha and David A Plaisted. On the mechanical derivation of loop invariants. *Journal of Symbolic Computation*, 15(5-6):705–744, 1993.

[9] YuTing Chen, Laura Kovács, and Simon Robillard. Theory-specific reasoning about loops with arrays using vampire. In *Proceedings of the 3rd Vampire Workshop*, pages 16–32. EasyChair, 2017.

[10] Michael A Colón, Sriram Sankaranarayanan, and Henny B Sipma. Linear invariant generation using non-linear constraint solving. In *International Conference on Computer Aided Verification*, pages 420–432. Springer, 2003.

[11] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 105–118. ACM, 2011.

[12] Simon Cruanes. Superposition with structural induction. In *International Symposium on Frontiers of Combining Systems*, pages 172–188. Springer, 2017.

[13] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[14] Isil Dillig, Thomas Dillig, and Alex Aiken. Fluid updates: Beyond strong vs. weak updates. In *European Symposium on Programming*, pages 246–266. Springer, 2010.

[15] Melvin Fitting and Richard L Mendelsohn. *First-order Modal Logic*, volume 277. Springer Science & Business Media, 2012.

[16] Cormac Flanagan and Shaz Qadeer. Predicate abstraction for software verification. In *Proceedings of the 29th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 191–202. ACM, 2002.

[17] Ashutosh Gupta and Andrey Rybalchenko. InvGen: An efficient invariant generator. In *International Conference on Computer Aided Verification*, pages 634–640. Springer, 2009.

[18] Arie Gurfinkel, Sharon Shoham, and Yuri Meshman. SMT-based verification of parameterized systems. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 338–348. ACM, 2016.

[19] Aleksandr Karbyshev, Nikolaj Bjørner, Shachar Itzhaky, Noam Rinetzky, and Sharon Shoham. Property-directed inference of universal invariants or proving their absence. *Journal of the ACM*, 64(1):7, 2017.

[20] Egor George Karpenkov and David Monniaux. Formula slicing: Inductive invariants from preconditions. In *Haifa Verification Conference*, pages 169–185. Springer, 2016.

[21] Abdelkader Kersani and Nicolas Peltier. Combining superposition and induction: A practical realization. In *International Symposium on Frontiers of Combining Systems*, pages 7–22. Springer, 2013.

[22] Laura Kovács and Simon Robillard. Reasoning about loops using vampire. In *Vampire Workshop*, pages 52–62. EasyChair, 2015.

[23] 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*, pages 260–270. ACM, 2017.

[24] Laura Kovács and Andrei Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In *International Conference on Fundamental Approaches to Software Engineering*, pages 470–485, 2009.

[25] Laura Kovács and Andrei Voronkov. First-order theorem proving and Vampire. In *International Conference on Computer Aided Verification*, pages 1–35. Springer, 2013.

[26] Shuvendu K Lahiri and Randal E Bryant. Constructing quantified invariants via predicate abstraction. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 267–281. Springer, 2004.

[27] Daniel Larraz, Enric Rodríguez-Carbonell, and Albert Rubio. SMT-based array invariant generation. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 169–188. Springer, 2013.

[28] Zhen Li. *Efficient and Generic Reasoning for Modal Logics*. PhD thesis, The University of Manchester, UK, 2008.

[29] Kenneth L McMillan. Quantified invariant generation using an interpolating saturation prover. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 413–427. Springer, 2008.

[30] Ernst-Rüdiger Olderog. *General equivalence of expressivity definitions using strongest postconditions resp. weakest preconditions*. Inst. für Informatik u. Prakt. Mathematik, Christian-Albrechts-Univ., 1980.

[31] Giles Reger, Nikolaj Bjørner, Martin Suda, and Andrei Voronkov. AVATAR modulo theories. In *2nd Global Conference on Artificial Intelligence*, pages 39–52. EasyChair, 2016.

[32] Giles Reger, Martin Suda, and Andrei Voronkov. Unification with abstraction and theory instantiation in saturation-based reasoning. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 3–22. Springer, 2018.

[33] Geoff Sutcliffe. The TPTP problem library and associated infrastructure. *Journal of Automated Reasoning*, 59(4):483–502, 2017.

[34] Daniel Wand. *Superposition: Types and Induction*. PhD thesis, Saarland University, Saarbrücken, Germany, 2017.