Loop invariants on demand

K. Rustan M. Leino [0] and Francesco Logozzo [1]

[0] Microsoft Research, Redmond, WA, USA
leino@microsoft.com
[1] École Polytechnique, Palaiseau, France
Francesco.Logozzo@polytechnique.edu

Manuscript KRML 155, 7 June 2005.

Abstract. This paper describes a sound technique that combines the precision of theorem proving with the loop-invariant inference of abstract interpretation. The loop-invariant computations are invoked on demand when the need for a stronger loop invariant arises, which allows a gradual increase in the level of precision used by the abstract interpreter. The technique generates loop invariants that are specific to a subset of a program's executions, achieving a dynamic and automatic form of value-based trace partitioning. Finally, the technique can be incorporated into a lemmas-on-demand theorem prover, where the loop-invariant inference happens after the generation of verification conditions.

0 Introduction

A central problem in reasoning about software is the infinite number of control paths and data values that a program's executions may give rise to. The general solution to this problem is to perform abstractions [8]. Abstractions include, for instance, predicates of interest on the data (as is done in predicate abstraction [17]) and summaries of the effects of certain control paths (like loop invariants [15, 21]). A trend that has emerged in the last decade is to start with coarse-grained abstractions and to refine these when the need arises (as used, for example, in predicate refinement [17, 2, 20], lemmas-on-demand theorem provers [12, 11, 4, 1, 24], and abstract-interpretation based verifiers [28]).

In this paper, we describe a technique that refines loop invariants on demand. In particular, the search for stronger loop invariants is initiated as the need for stronger loop invariants arises during a theorem prover's attempt at proving the program. The technique can generate loop invariants that are specific to a subset of a program's executions, achieving a dynamic and automatic form of value-based trace partitioning. Finally, the technique can be incorporated into a lemmas-on-demand theorem prover, where the loop-invariant inference happens after the generation of verification conditions.

The basic idea is this: Given a program, we generate a *verification condition*, a logical formula whose validity implies the correctness of the program. We pass this formula to an automatic theorem prover that will either prove the correctness of the program or produce, in essence, a set of candidate program traces that lead to an error. Rather than

```
x := 0 ;  m := 0 ;
while (x < N) {
  if (...) {          /* check if a new minimum as been found */
    m := x ;
  }
  x := x + 1 ;
}
if (0 < N) {
  assert 0 ⩽ m < N ;
}
```

Fig. 0. A running example, showing a correct program whose correctness follows from a disjunctive loop invariant. The program is an abstraction of a program that iterates through an array (not shown) of length N (indexed from 0), recording in m the index of the array's mimimum element. The then branch of the if statement after the loop represents some operation that relies on the fact that m is a proper index into the array.

just giving up and reporting these candidate traces as an error, we invoke an abstract interpreter on the loops along the traces, hoping to find stronger loop invariants that will allow the theorem prover to make more progress toward a proof. As this process continues, increasingly more precise analyses and abstract domains may be used with the abstract interpreter, which allows the scheme of optimistically trying cheaper analyses first. Each invocation of the abstract interpreter starts from the information available in the candidate traces. Consequently, the abstract interpreter can confine its analysis to these traces, thus computing loop invariants that hold along these traces but that may not hold for all of the program's executions. Once loop invariants are computed, they are communicated back to the theorem prover. This process terminate when the theorem prover is able to prove the program correct or when the abstract interpreter runs out of steam, in which case the candidate traces are reported.

As an example, consider the program in Figure 0. The strongest invariant for the loop is:

$$(N \leqslant 0 \wedge x = m = 0) \vee (0 < N \wedge 0 \leqslant x \leqslant N \wedge 0 \leqslant m < N)$$

Using this loop invariant, one can prove the program to be correct, that is, one can prove that the assertion near the end of the program never fails. However, because this invariant contains a disjunction, common abstract domains like intervals [8], octagons [29], and polyhedra [10] would not be able to infer it. The strongest loop invariant that does not contain a disjunction is:

$$0 \leqslant x \wedge 0 \leqslant m$$

which, disappointingly, is not strong enough to prove the program correct. Disjunctive completion [9] or loop unrolling can be used to improve the precision of these domains, enough to prove the property. Nevertheless, in practice the cost of disjunctive completion is prohibitive and in the general case the number of loop unrollings necessary to prove the properties of interest is not known.

Trace partitioning is a well-known technique that provides a good value of the precision-to-cost ratio for handling disjunctive properties. Our technique performs trace partitioning dynamically (*i.e.*, during the analysis of the program), automatically, (*i.e.*, without requiring interaction from the user), and contextually (*i.e.*, according to the values of variables and the control flow of the program).

Applying our technique to the example program in Figure 0, the theorem prover would first produce a set of candidate traces that exit the loop with any values for x and m, then takes the then branch of the if statement (which implies $0 < N$) and then finds the assertion to be false. Not all such candidate trace are feasible, however. From the information about the candidate traces, and in particular from $0 < N$, the abstract interpreter (with the octagon or polyhedra domain, for example) infers the loop invariant:

$$0 < N \wedge 0 \leqslant x \leqslant N \wedge 0 \leqslant m < N$$

using which the theorem prover can complete the proof.

In Section 1, we present a toy imperative language and define an abstract interpretation for it, parameterized by any abstract domain. We also prescribe for this language the generation of verification conditions. In Section 2, we present the basic interface of the theorem prover and define our technique as a "fact generator" for the theorem prover. Throughout, we apply what we say to the running example shown in Figure 0. We relate our technique to previous work in Section 3 and conclude the paper in Section 4.

1 An imperative language and its semantics

In this section, we define a while language, its abstract semantics, and its associated verification conditions. We also show an example program whose analysis benefits from the combination of an abstract interpreter and a theorem prover.

1.0 Grammar

We consider the source language whose grammar is given in Figure 1. The source language includes support for specifications via the $\mathbf{assert}\ E$ statement: if the expression E evaluates to false, then the program fails. The assignment statement $x := E$ sets the variable x to the value of the expression E. The havoc statement $\mathbf{havoc}\ x$ nondeterministically assigns a value to x. Sequential composition, conditionals, and loops are the usual ones. Note that we assume that loops are uniquely determined by labels ℓ taken from a set $\mathcal{W}$: Given a label ℓ, the function $\mathsf{LookupWhile}(\ell)$ returns the while loop associated with such a label.

For example, Figure 2 shows the program in Figure 0 is written in the notation of our source language. Note the use of $\mathbf{havoc}\ b;$ followed by a use of b in a conditional, which encodes an arbitrary choice between the two branches of the conditional.

$$
\begin{array}{lll}
\mathit{Stmt} ::= & \mathbf{assert}\ \mathit{Expr}; & \text{(assertion)} \\
\mid & x := \mathit{Expr}; & \text{(assignment)} \\
\mid & \mathbf{havoc}\ x; & \text{(set } x \text{ to an arbitrary value)} \\
\mid & \mathit{Stmt}\ \mathit{Stmt} & \text{(composition)} \\
\mid & \mathbf{if}\ (\mathit{Expr})\ \{\mathit{Stmt}\}\ \mathbf{else}\ \{\mathit{Stmt}\} & \text{(conditional)} \\
\mid & \mathbf{while}^{\ell}\ (\mathit{Expr})\ \{\mathit{Stmt}\} & \text{(loop)}
\end{array}
$$

Fig. 1. The grammar of the source language.

$$
\begin{array}{l}
x := 0;\ \ m := 0; \\
\mathbf{while}^{\ell}\ (x < N)\ \{ \\
\quad \mathbf{havoc}\ b;\ \ \mathbf{if}\ (b)\ \{m := x;\}\ \mathbf{else}\ \{\mathbf{assert}\ \mathit{true};\} \\
\quad x := x + 1; \\
\} \\
\mathbf{if}\ (0 < N)\ \{\mathbf{assert}\ 0 \leqslant m < N;\}\ \mathbf{else}\ \{\mathbf{assert}\ \mathit{true};\}
\end{array}
$$

Fig. 2. The program of Figure 0 encoded in the source language.

1.1 Abstract semantics

The abstract semantics $\mathbb{c}[\![\cdot]\!]$ is defined by structural induction in Figure 3. It is parameterized by an abstract domain $\mathcal{D}$, which includes a join operator ($\sqcup$), a meet operator ($\sqcap$), a projection operator (eliminate), and a primitive to handle the assignment (assign). The pointwise extension of the join is denoted $\dot{\sqcup}$.

The abstract semantics for expressions is given by a function $\mathbb{b}[\![\cdot]\!] \in \mathcal{L}(\mathit{Expr}) \to \mathcal{D} \to \mathcal{D}$. Intuitively, $\mathbb{b}[\![\mathrm{E}]\!](d)$ overapproximates the set of concrete states $\gamma(d)$ that makes the expression E true. For lack of space, we omit here its definition and we refer the interest reader to, *e.g.* , [7].

The input of the abstract semantics is an abstract state representing the initial conditions. The output is a pair consisting of an approximation of the output states and a map from (the label of) each loop sub-statement to an approximation of its loop invariant. The rules in Figure 3 are described as follows. The **assert** statement retains the part of the input state that satisfy the asserted expression. The effects of an assignment are handled by the abstract domain through the primitive assign. In the concrete, **havoc** x sets the variable x to any value, so in the abstract we handle it by simply projecting out the variable x from the abstract state. Stated differently, we set the value of x to $\top$. Sequential composition, conditional, and loops are defined as usual. In particular, the semantics of a loop is given by a least fixpoint on the abstract domain $\mathcal{D}$. Such a fixpoint can be computed iteratively, and if the abstract domain does not satisfy the ascending chain condition then the convergence (to a post-fixpoint) of the iterations is enforced through the use of a widening operator. The abstract state just after the loop is given by the loop invariant restrainted by the negation of the guard. Notice that the abstract semantics for the loop also records in the output map the loop invariant for ℓ.

$$
\begin{array}{l}
\mathbb{c}[\![\cdot]\!] \in \mathcal{L}(\mathit{Stmt}) \to \mathcal{D} \to \mathcal{D} \times (\mathcal{W} \to \mathcal{D}) \\
\mathbb{c}[\![\textbf{assert}\ \mathrm{E};]\!](d) \ = \ (\mathbb{b}[\![\mathrm{E}]\!](d),\ \emptyset) \\
\mathbb{c}[\![x := \mathrm{E};]\!](d) \ = \ (d.\mathsf{assign}(x, \mathrm{E}),\ \emptyset) \\
\mathbb{c}[\![\textbf{havoc}\ x;]\!](d) \ = \ (d.\mathsf{eliminate}(x),\ \emptyset) \\
\mathbb{c}[\![\mathrm{S}_0\ \mathrm{S}_1]\!](d) \ = \ \textbf{let}\ (d_0, f_0) = \mathbb{c}[\![\mathrm{S}_0]\!](d)\ \textbf{in} \\
\qquad\qquad\qquad\qquad \textbf{let}\ (d_1, f_1) = \mathbb{c}[\![\mathrm{S}_1]\!](d_0)\ \textbf{in} \\
\qquad\qquad\qquad\qquad (d_1,\ f_0 \cup f_1) \\
\mathbb{c}[\![\textbf{if}\ (\mathrm{E})\ \{\mathrm{S}_0\}\ \textbf{else}\ \{\mathrm{S}_1\}]\!](d) \ = \ \textbf{let}\ (d_0, f_0) = \mathbb{c}[\![\mathrm{S}_0]\!](\mathbb{b}[\![\mathrm{E}]\!](d)\ \textbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{let}\ (d_1, f_1) = \mathbb{c}[\![\mathrm{S}_1]\!](\mathbb{b}[\![\neg \mathrm{E}]\!](d)\ \textbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (d_0 \sqcup d_1,\ f_0 \cup f_1) \\
\mathbb{c}[\![\textbf{while}^{\ell}\ (\mathrm{E})\ \{\mathrm{S}\}]\!](d) \ = \\
\qquad \textbf{let}\ (d^*, f^*) = \mathsf{lfp}(\ \lambda\, X,\ Y \ \bullet \ (d, \emptyset)\ \dot{\sqcup}\ \mathbb{c}[\![\mathrm{S}]\!](\mathbb{b}[\![\mathrm{E}]\!](X))\ )\ \textbf{in} \\
\qquad (\mathbb{b}[\![\neg \mathrm{E}]\!](d^*),\ f^*[\ell \mapsto d^*])
\end{array}
$$

Fig. 3. The generic abstract semantics for the source language.

$$
\begin{array}{lll}
\mathit{Cmd} ::= & \textbf{assert}\ \mathit{Expr} & \text{(assert)} \\
\mid & \textbf{assume}\ \mathit{Expr} & \text{(assume)} \\
\mid & \mathit{Cmd}\ ;\ \mathit{Cmd} & \text{(sequence)} \\
\mid & \mathit{Cmd}\ \square\ \mathit{Cmd} & \text{(non-deterministic choice)}
\end{array}
$$

Fig. 4. The intermediate language.

1.2 Verification conditions

To define the verification conditions for programs written in our source language, we first translate them into an intermediate language and then apply weakest preconditions (*cf.* [25]).

Intermediate language The commands of the intermediate language are given by the grammar in Figure 4. Our intermediate language is that of passive commands, *i.e.*, assignment and loop-free commands [14].

The **assert** and the **assume** statements first evaluate the expression Expr. If it evaluates to true, then the execution continues. If the expression evaluates to false, the **assert** statement causes the program to fail (the program *goes wrong*) and the **assume** statement blocks the program (which implies that the program has no chance of going wrong). Furthermore, we have a statement for sequential composition and non-deterministic choice.

The translation from a source language program S to an intermediate language program is given by the following function (**id** denotes the identity map):

$$\mathsf{translate}(\mathrm{S}) \ = \ \textbf{let}\ (\mathrm{C}\ , m) = \mathsf{tr}(\mathrm{S}\ , \mathbf{id})\ \textbf{in}\ \mathrm{C}$$

The goals of the translation is to get rid of (i) assignments and (ii) loops. To achieve (i), the translation uses a variant of static single assignment (SSA) [0] that introduces new variables (*inflections*) that stand for the values of program variables at different

source-program location and that within any one execution path has only one value. To achieve (ii), the translation replaces an arbitrary number of iterations of a loop with something that describes the effect that these iterations have on the variables, namely the loop invariant. The definition of the function tr is in Figure 5. The function takes as input a program in the source language and a renaming function from program variables to their pre-state inflections, and it returns a program in the intermediate language and a renaming function from program variables to their post-state inflections. The rules in Figure 5 are described as follows.

The translation of an $\mathbf{assert}$ just renames the variables in the asserted expression to their current inflections. One of the goals of the passive form is to get rid of the assignments. As a consequence, given an assignment $x := \mathrm{E}$ in the source language, we generate a fresh variable for x (intuitively, the value of x after the assignment), we apply the renaming function to E, and we output an $\mathbf{assume}$ statement that binds the new variable to the renamed expression. For instance, a statement that assigns to y in a state where the current inflection of y is translated as follows, where $y1$ is a fresh variable that denotes the inflection of y after the statement:

$$\mathsf{tr}(y := y + 4,\ [y \mapsto y0]) \ = \ (\mathbf{assume}\ y1 = y0 + 4,\ [y \mapsto y1])$$

The translation of $\mathbf{havoc}\ x$ just binds x to a fresh variable, without introducing any assumptions about the value of this fresh variable. The translation of sequential composition yields the composition of the translated statements and the post-renaming of the second statement. The translations of the conditional and the loop are more tricky.

For the conditional, we translate the two branches to obtain two translated statements and two renaming functions. Then we consider the set of all the variables on which the renaming functions disagree (intuitively, they are the variables modified in one or both the branches of the conditional), and we assign them fresh names. These names will be the inflections of the variables after the conditional statement. Then, we generate the translation of the true (resp. false) branch of the conditional by assuming the guard (resp. the negation of the guard), followed by the translated command and the assumption of the fresh names for the modified variables. Finally, we use the non-deterministic choice operator to complete the translation of the whole conditional statement.

For the loop, we first identify the loop targets (defined in Figure 6), generate fresh names for them, and translate the loop body. Then, we generate a fresh predicate symbol indexed by the loop identifier (J_ℓ), which intuitively stands for the invariant of the loop $\mathsf{LookupWhile}(\ell)$. We output a sequence that is made up by the assumption of the loop invariant (intuition: we have performed an arbitrary number of loop iterations) and a non-deterministic choice between two cases: (i) the loop condition evaluates to true, we execute a further iteration of the body, and then we stop checking ($\mathbf{assume}\ \mathit{false}$), or (ii) the loop condition evaluates to false and we terminate normally. Finally, please note that the arguments of the loop-invariant predicate J include the names of program variables at the beginning of the loop ($\mathsf{range}(m)$), and the names of the variables after an arbitrary number of iterations of the loop ($\mathsf{range}(n)$).

Please note that we tacitely assume a total order on variables, so that, *e.g.* , the sets $\mathsf{range}(m)$ and $\mathsf{range}(n)$ can be isomorphically represented as lists of variables. We will use the list representation in our examples.

$$\mathsf{tr} \in \mathcal{L}(Stmt) \times (\mathtt{Vars} \to \mathtt{Vars}) \to \mathcal{L}(Cmd) \times (\mathtt{Vars} \to \mathtt{Vars})$$

$\mathsf{tr}(\mathbf{assert}\ \mathrm{E};,\ m)\ =\ (\mathbf{assert}\ m(\mathrm{E}),\ m)$
$\mathsf{tr}(\mathtt{x} := \mathrm{E};,\ m)\ =\ (\mathbf{assume}\ \mathtt{x}' = m(\mathrm{E}),\ m[\mathtt{x} \mapsto \mathtt{x}'])$ where $\mathtt{x}'$ is a fresh variable
$\mathsf{tr}(\mathbf{havoc}\ \mathtt{x};,\ m)\ =\ (\mathbf{assume}\ \mathit{true},\ m[\mathtt{x} \mapsto \mathtt{x}'])$ where $\mathtt{x}'$ is a fresh variable
$\mathsf{tr}(\mathrm{S}_0\ \mathrm{S}_1,\ m)\ =\ \mathbf{let}\ (\mathrm{C}_0, n_0) = \mathsf{tr}(\mathrm{S}_0, m)\ \mathbf{in}$
$\qquad\qquad\qquad\qquad\ \mathbf{let}\ (\mathrm{C}_1, n_1) = \mathsf{tr}(\mathrm{S}_1, n_0)\ \mathbf{in}$
$\qquad\qquad\qquad\qquad\ (\mathrm{C}_0\ ;\ \mathrm{C}_1,\ n_1)$
$\mathsf{tr}(\mathbf{if}\ (\mathrm{E})\ \{\mathrm{S}_0\}\ \mathbf{else}\ \{\mathrm{S}_1\},\ m)\ =$
$\quad \mathbf{let}\ (\mathrm{C}_0, n_0) = \mathsf{tr}(\mathrm{S}_0, m)\ \mathbf{in}$
$\quad \mathbf{let}\ (\mathrm{C}_1, n_1) = \mathsf{tr}(\mathrm{S}_1, m)\ \mathbf{in}$
$\quad \mathbf{let}\ \mathtt{V} = \{\mathtt{x} \in \mathtt{Vars} \mid n_0(\mathtt{x}) \neq n_1(\mathtt{x})\}\ \mathbf{in}$
$\quad \mathbf{let}\ \mathtt{V}'\ \mathbf{be}$ fresh variables for the variables in $\mathtt{V}\ \mathbf{in}$
$\quad \mathbf{let}\ \mathrm{D}_0 =\ \mathbf{assume}\ m(\mathrm{E})\ ;\ \mathrm{C}_0\ ;\ \mathbf{assume}\ \mathtt{V}' = n_0(\mathtt{V})\ \mathbf{in}$
$\quad \mathbf{let}\ \mathrm{D}_1 =\ \mathbf{assume}\ \neg m(\mathrm{E})\ ;\ \mathrm{C}_1\ ;\ \mathbf{assume}\ \mathtt{V}' = n_1(\mathtt{V})\ \mathbf{in}$
$\quad (\mathrm{D}_0 \,\square\, \mathrm{D}_1,\ m[\mathtt{V} \mapsto \mathtt{V}'])$
$\mathsf{tr}(\mathbf{while}^{\ell}\ (\mathrm{E})\ \{\mathrm{S}\},\ m)\ =$
$\quad \mathbf{let}\ \mathtt{V} = \mathsf{targets}(\mathrm{S})\ \mathbf{in}$
$\quad \mathbf{let}\ \mathtt{V}'\ \mathbf{be}$ fresh variables for the variables in $\mathtt{V}\ \mathbf{in}$
$\quad \mathbf{let}\ n = m[\mathtt{V} \mapsto \mathtt{V}']\ \mathbf{in}$
$\quad \mathbf{let}\ (\mathrm{C}, n_0) = \mathsf{tr}(\mathrm{S}, n)\ \mathbf{in}$
$\quad \mathbf{let}\ J_\ell\ \mathbf{be}$ a fresh predicate symbol $\mathbf{in}$
$\quad (\mathbf{assume}\ J_\ell\langle \mathsf{range}(m),\ \mathsf{range}(n)\rangle\ ;$
$\quad\ (\mathbf{assume}\ n(\mathrm{E})\ ;\ \mathrm{C}\ ;\ \mathbf{assume}\ \mathit{false}\ \ \square\ \ \mathbf{assume}\ \neg n(\mathrm{E})),\ \ n)$

Fig. 5. The function that translates from the source program to our intermediate language.

For example, applying translate to the program in Figure 2 results in the intermediate-language program shown in Figure 7.

Weakest preconditions The weakest preconditions of a program in the intermediate language are given in Figure 8, where Φ denotes the set of first-order formulae. They characterize the all the pre-states from which every non-blocking execution of the command does not go wrong, and from which every terminating execution ends in a state satisfying Q. As a consequence, the verification condition for a given program S, in the source language, is

$$\mathsf{wp}(\mathsf{translate}(\mathrm{S}), \mathit{true}) \tag{0}$$

For example, the verification condition for the source program in Figure 2, obtained as the weakest precondition of the intermediate-language program in Figure 7, is the formula shown in Figure 9. (As can be seen in this formula, the verification condition contains a noticeable amount of redundancy, even for this small source program. We don't show it here, but the redundancy can be eliminated by using an important optimization in the computation of weakest preconditions, which is enabled by the fact that the weakest preconditions are computed from passive commands, see [14, 23].)

$$
\begin{array}{l}
\mathsf{targets} \in \mathcal{L}(\mathit{Stmt}) \to \mathcal{P}(\mathtt{Vars}) \\
\mathsf{targets}(\textbf{assert}\ \mathrm{E};) \ = \ \emptyset \\
\mathsf{targets}(\mathtt{x} := \mathrm{E};) \ = \ \mathsf{targets}(\textbf{havoc}\ \mathtt{x};) \ = \ \{\mathtt{x}\} \\
\mathsf{targets}(\mathrm{S}_0\ \mathrm{S}_1) \ = \ \mathsf{targets}(\textbf{if}\ (\mathrm{E})\ \{\mathrm{S}_0\}\ \textbf{else}\ \{\mathrm{S}_1\}) \ = \ \mathsf{targets}(\mathrm{S}_0) \cup \mathsf{targets}(\mathrm{S}_1) \\
\mathsf{targets}(\textbf{while}^{\ell}\ (\mathrm{E})\ \{\mathrm{S}\}) \ = \ \mathsf{targets}(\mathrm{S})
\end{array}
$$

Fig. 6. The assignment targets, that is, the set of variables assigned in a source statement.

$$
\begin{array}{l}
\textbf{assume}\ x_0 = 0\ ;\ \textbf{assume}\ m_0 = 0\ ; \\
\textbf{assume}\ J_\ell\langle (x_0, m_0, b, N),\ (x_1, m_1, b_0, N)\rangle\ ; \\
(\ \ \textbf{assume}\ x_1 < N\ ; \\
\quad (\ \ \textbf{assume}\ b_1\ ;\ \textbf{assume}\ m_2 = x_1\ ;\ \textbf{assume}\ m_3 = m_2 \\
\quad \Box\ \textbf{assume}\ \neg b_1\ ;\ \textbf{assert}\ \mathit{true}\ ;\ \textbf{assume}\ m_3 = m_1 \\
\quad)\ ; \\
\quad \textbf{assume}\ x_2 = x_1 + 1\ ;\ \textbf{assume}\ \mathit{false} \\
\Box \\
\quad \textbf{assume}\ \neg(x_1 < N) \\
)\ ; \\
(\ \ \textbf{assume}\ 0 < N\ ;\ \textbf{assert}\ 0 \leqslant m_1 < N \\
\Box\ \textbf{assume}\ \neg(0 < N)\ ;\ \textbf{assert}\ \mathit{true} \\
)
\end{array}
$$

Fig. 7. The intermediate-language program obtained as a translation of the source-language program in Figure 2. J_ℓ is a predicate symbol corresponding to the loop.

1.3 The benefit of combining analysis techniques

It is unreasonable to think that every static analysis technique would encode all details of the operators (like integer addition, bitwise-or, and floating-point division) in a programming language. Operators without direct support can be encoded as uninterpreted functions. A theorem prover that supports quantifications offers an easy way to encode interesting properties of such functions. For example, the property that the bitwise-or of two non-negative integers is non-negative can be added to the analysis simply by including

$$(\ \forall\, x, y \bullet\ 0 \leqslant x \ \wedge\ 0 \leqslant y\ \Rightarrow\ 0 \leqslant \mathit{bitwiseOr}(x, y)\ ) \tag{1}$$

in the antecedent of the verification condition. In an abstract interpreter, the addition of properties like this requires authoring or modifying an abstract domain, which takes more effort. On the other hand, to use the theorem prover to prove a program correct, one needs to supply it with inductive conditions like invariants. An abstract interpreter computes (over-approximations of) such invariants. By combining an abstract interpreter and a theorem prover, one can reap the benefits of both the abstract interpreter's invariant computation and the theorem prover's high precision and easy extensibility.

For example, consider the program in Figure 10, where we use "$|$" to denote bitwise-or. Without a loop invariant, the theorem prover cannot prove that the first assertion

$$\mathsf{wp} \in \mathcal{L}(Cmd) \times \Phi \to \Phi$$
$$\mathsf{wp}(\mathbf{assert}\ \mathrm{E}\,,\ Q) \ =\ \mathrm{E} \ \wedge\ Q$$
$$\mathsf{wp}(\mathbf{assume}\ \mathrm{E}\,,\ Q) \ =\ \mathrm{E} \ \Rightarrow\ Q$$
$$\mathsf{wp}(\mathrm{C}_0\ ;\ \mathrm{C}_1,\ Q) \ =\ \mathsf{wp}(\mathrm{C}_1,\ \mathsf{wp}(\mathrm{C}_0, Q))$$
$$\mathsf{wp}(\mathrm{C}_0 \,\square\, \mathrm{C}_1,\ Q) \ =\ \mathsf{wp}(\mathrm{C}_0, Q) \wedge \mathsf{wp}(\mathrm{C}_1, Q)$$

Fig. 8. Weakest preconditions of the intermediate language.

$$\begin{array}{l}
x_0 = 0 \ \Rightarrow\ m_0 = 0 \ \Rightarrow \\
J_\ell\langle (x_0, m_0, b, N),\ (x_1, m_1, b_0, N)\rangle \ \Rightarrow \\
(x_1 < N \ \Rightarrow \\
\quad (b_1 \ \Rightarrow\ m_2 = x_1 \ \Rightarrow\ m_3 = m_2 \ \Rightarrow\ x_2 = x_1 + 1 \ \Rightarrow\ \mathit{false} \ \Rightarrow\ \ldots) \wedge \\
\quad (\neg b_1 \ \Rightarrow\ \mathit{true} \wedge (m_3 = m_1 \ \Rightarrow\ x_2 = x_1 + 1 \ \Rightarrow\ \mathit{false} \ \Rightarrow\ \ldots)) \\
) \wedge \\
(\neg(x_1 < N) \ \Rightarrow \\
\quad (0 < N \ \Rightarrow\ 0 \leqslant m_1 < N \wedge \mathit{true}) \wedge \\
\quad (\neg(0 < N) \ \Rightarrow\ \mathit{true} \wedge \mathit{true}) \\
)
\end{array}$$

Fig. 9. The weakest precondition of the program in Figure 7. We use $\Rightarrow$ as a right-associative operator with lower precedence than $\wedge$. The ellipsis in each of the two occurrences of the sub-formula "$\mathit{false} \ \Rightarrow\ \ldots$" stands for the conjunction shown in the second and third last lines.

holds. Without support for bitwise-or, an abstract interpreter cannot prove it either. But the combination can prove it: an abstract interpreter with support for intervals infers the loop invariant $0 \leqslant x$, and given this loop invariant and the axiom (1), a theorem prover can prove the program correct.

2 Loop-Invariant Fact Generator

To determine whether or not a program is correct with respect to its specification, we need to determine the validity of the verification condition (0), which we do using a theorem prover. A theorem prover can equivalently be thought of as a satisfier, since a formula is valid if and only if its negation is unsatisfiable. In this paper, we take the view of the theorem prover being a satisfier, so we ask it to try to satisfy the formula $\neg(0)$. If the theorem prover's exhaustive search determines that $\neg(0)$ is unsatisfiable, then (0) is valid and the program is correct. Otherwise, the prover returns a *monome*—a conjunction of possibly negated atomic formulas—that satisfies $\neg(0)$ and, as far as the prover can tell, is consistent. Intuitively, the monome represents a set of execution paths that lead to an error in the program being analyzed, together with any information gathered by the theorem prover about these execution paths.

A satisfying monome returned by the theorem prover may be an indication of an actual error in the program being analyzed. But the monome may also be the result of too weak loop invariants. (There's a third possibility: that the program's correctness depends on mathematical properties that are beyond the power or resource bounds of

```
x := 0 ;
while^ℓ (x < N) {
  if (0 ⩽ y) {assert 0 ⩽ x | y;} else {assert true;}
  x := x + 1;
}
```

Fig. 10. An artificial program whose analysis benefits from the combination of an abstract interpreter and a theorem prover.

the prover. In this paper, we offer no improvement for this third possibility.) At the point where the prover is about to return a satisfying monome, we would like a chance to infer stronger loop invariants. To explain how this is done, let us give some more details of the theorem prover.

We assume the architecture of a lemmas-on-demand theorem prover [12, 11]. It starts off viewing the given formula as a propositional formula, with each atomic subformula being represented as a propositional variable. The theorem prover asks a boolean-satisfiability (SAT) solver to produce monomes that satisfy the formula propositionally. Each such monome is then scrutinized by the supported *theories*. These theories may include, for example, the theory of uninterpreted function symbols with equality and the theory of linear arithmetic. If the monome is found to be inconsistent with the theories, the theories generate a *lemma* that explains the inconsistency. The lemma is then, in effect, conjoined with the original formula and the search for a satisfying monome continues.

If the SAT solver finds a monome that is consistent with the theories, the theorem prover invokes a number of *fact generators* [24], each of which is allowed to return facts that may help refute the monome. For example, one such fact generator is the quantifier module, which Skolemizes existential quantifiers and heuristically instantiates universal quantifiers [13, 24]. Unlike the lemmas generated by the theories, the facts may or may not be helpful in refuting the monome. Any facts generated are taken into consideration and the search for a satisfying monome is resumed. Only if the fact generators have no further facts to produce, or if some limit on the number of fact-generator invocations has been reached, does the theorem prover return the satisfying monome.

To generate loop invariants on demand, we therefore build a fact generator that infers loop invariants for the loops that play a role in the current monome. This fact generator can apply more powerful technique with each invocation. For example, in a subsequent invocation, the fact generator may make use of more detailed abstract domains, it may perform more join operations before applying accelerating widen operations, or it may apply more narrowing operations. The fact generator can also make use of the contextual information of the current monome when inferring loop invariants—a loop invariant so inferred may not be a loop invariant in every execution of the program, but it may be good enough to refute the current monome.

The routine that generates new loop-invariant facts is shown in Figure 11. The GenerateFacts routine extracts from the monome each loop-invariant predicate $J_\ell\langle \mathtt{V}_0, \mathtt{V}_1\rangle$ of interest. The inference of a loop invariant for loop ℓ is done as follows.

```
GenerateFacts(Monome μ) =
  let V be the program variables in
  var Facts := ∅ ;
  foreach J_ℓ⟨V_0, V_1⟩ ∈ μ {
    var d_0 := α(μ) ;
    foreach x ∉ V_0 { d_0 := d_0.eliminate(x); }
    var d := d_0 ⊓ 𝔟[[V = V_0]](d_0) ;
    let (_, f) = [[LookupWhile(ℓ)]](d) in
    let m = V → V_1 in
    let LoopInv = m(f(ℓ)) in
      Facts := Facts ∪ {γ(d_0) ∧ J_ℓ⟨V_0, V_1⟩ ⇒ γ(LoopInv)} ;
  }
  return Facts
```

Fig. 11. The GenerateFacts routine, which invokes the abstract intepreter to infer loop invariants.

First, GenerateFacts computes into d an initial state for the loop ℓ. This initial state is computed as a projection of the monome μ onto the loop pre-state inflections ($\mathtt{V}_0$). We let the abstract interpreter compute this projection, so we start by applying the abstraction function α, which maps the monome to an abstract element. For instance, using the polyhedra abstract domain, the α keeps just the parts of the monome that involve linear inequalities, all the other parts being abstracted away. The initial state, which is in terms of $\mathtt{V}_0$, is then conjoined with a set of equalities such that each program variable in $\mathtt{V}$ has the value of the corresponding variable in $\mathtt{V}_0$.

Then, GenerateFacts fetches the loop to be analyzed (LookupWhile(ℓ)) and runs the abstract interpreter with the initial state d. Since the abstract element d represents a relation on $\mathtt{V}_0$ and $\mathtt{V}$, the analysis will infer a relational invariant [26].

The abstract interpreter returns a loop invariant $f(\ell)$ with variables in $\mathtt{V}$ and $\mathtt{V}_0$. The routine GenerateFacts then renames each *program* variable (in $\mathtt{V}$) to its corresponding loop post-state inflection (in $\mathtt{V}_1$). Finally, the set of gathered facts is updated with the implication

$$\gamma(d) \wedge J_\ell\langle \mathtt{V}_0,\ \mathtt{V}_1\rangle \Rightarrow \gamma(\mathit{LoopInv})$$

Intuitively, it says that if the execution trace being examined satisfies d—the initial state of the loop used in the analysis just performed—then the loop-invariant predicate $J_\ell\langle \mathtt{V}_0,\ \mathtt{V}_1\rangle$ is no weaker than the inferred invariant $\mathit{LoopInv}$. In this formula, the abstract domain elements d and $\mathit{LoopInv}$ are first concretized using the concretization function γ, which produces a first-order formula in Φ.

Not utilized in Figure 11 are the invariants inferred for nested loops, which are also recorded in f. With a little more bookkeeping (namely, keeping track of the pre- and post-state inflections of the nested loop), one can also produce facts about these loops.

Continuing our example, when the negation of the verification condition in Figure 9 is passed to the theorem prover, the theorem prover responds with the following monome:

$$x_0 = 0 \wedge m_0 = 0 \wedge$$
$$J_\ell\langle(x_0, m_0, b, N),\ (x_1, m_1, b_0, N)\rangle \wedge$$
$$\neg(x_1 < N) \wedge$$
$$0 < N \wedge \neg(0 \leqslant m_1)$$

(or, depending on how the SAT solver makes its case splits, the last literal in the monome may instead be $\neg(m_1 < N)$). When this monome is passed to GenerateFacts in Figure 11, the routine finds the $J_\ell\langle(x_0, m_0, b, N),\ (x_1, m_1, b_0, N)\rangle$ predicate, which tells it to analyze loop ℓ. We assume for this example that a numeric abstract domain like the polyhedra domain [10] is used. Since loop ℓ's pre-state inflections are (x_0, m_0, b, N), the abstract element d_0 is computed as

$$x_0 = 0 \wedge m_0 = 0 \wedge 0 < N$$

and d thus becomes

$$x_0 = 0 \wedge m_0 = 0 \wedge 0 < N \wedge$$
$$x_0 = x \wedge m_0 = m \wedge b = b \wedge N = N$$

The analysis of the loop produces in $f(\ell)$ the loop invariant

$$x_0 = 0 \leqslant x \leqslant N \wedge m_0 = 0 \leqslant m < N$$

Note that this loop invariant does not hold in general—it only holds in those executions where $0 < N$. Interestingly enough, notice that the condition $0 < N$ occurs *after* the loop in the program, but since it is relevant to the candidate error trace, it is part of the monome and thus becomes considered during the inference of loop invariants. Finally, the program variables are renamed to their post-state inflections (to match the second set of arguments passed to the J_ℓ predicate), which yields

$$x_0 = 0 \leqslant x_1 \leqslant N \wedge m_0 = 0 \leqslant m_1 < N$$

and so the generated fact is

$$x_0 = 0 \wedge m_0 = 0 \wedge 0 < N \wedge J_\ell\langle(x_0, m_0, b, N),\ (x_1, m_1, b_0, N)\rangle \Rightarrow$$
$$x_0 = 0 \leqslant x_1 \leqslant N \wedge m_0 = 0 \leqslant m_1 < N$$

With this fact as an additional constraint, the theorem prover is not able to satisfy the given formula. Hence, the program in Figure 2 has been automatically proved to be correct.

3 Related Work

Handjieva and Tzolovski [18] introduced a trace partitioning technique based on a program's control points. Their technique augments abstract states with an encoding of the history of the control flow. They consider finite sequences over $\{t_i, f_i\}$, where i is a

control point and t_i (resp. f_i) denotes the fact that the true (resp. false) branch at the control point i is taken. Nevertheless, their approach abstracts away from the values of variables at control points, so that with such a technique it is impossible to prove correct the example in Figure 0.

The trace partitioning technique used by Jeannet *et al.* [22], allows nperforming the partition according the values of boolean variables or linear constraints appearing in the program text. Their technique is effective for the analysis of reactive systems, but in the general case it suffers from being too syntactic-based.

Bourdoncle considers a form of dynamic partitioning [5] for the analysis of recursive functions. In particular, he refines the solution of representing disjunctive properties by a set of abstract elements (roughly, the disjuctive completion of the underlying abstract domain), by limiting the number of disjuncts through the use of a suitable widening operator.

Mauborgne and Rival [28] present a systematic view of trace partitioning techniques, and in particular they focus on automatic partitioning for proving the absence of run-time errors in large critical embedded software. The main difference with our work is that in our case the partition is driven by the property to be verified, *i.e.*, the refuted verification condition.

Finally, the theoretical bases for trace partitioning are given by the reduced cardinal product (RDC) of abstract domains [9, 16]. Roughly, the RDC of two abstract domains $\mathcal{D}_0$ and $\mathcal{D}_1$ produces a new domain made up of all the monotonic functions with domain $\mathcal{D}_0$ and co-domain $\mathcal{D}_1$. In trace partitioning, $\mathcal{D}_0$ contains the elements that allow the partitioning.

Yorsh *et al.* [31] and Zee *et al.* [32] use approaches different from ours for combining theorem proving and abstract interpretation: The first rely on a theorem prover to compute the best abstract transfer function for shape analysis. The second use an interactive theorem prover to verify that some program parts satisfy their specification. Then, a static analysis that assumes the specifications is run on the whole program to prove its correctness.

Henzinger *et al.* [20, 19] use the proof of unsatisfiability produced by a theorem prover to systematically reduce the abstract models checked by the BLAST model checker. The technique used in BLAST has several intriguing similarities to our technique. Two differences are that (i) our analysis is mainly performed inside the theorem prover, whereas BLAST is a separate tool built around a model checker, and (ii) our technique uses widening, whereas BLAST uses Craig's interpolation.

4 Conclusion

We have presented a technique that combines the precision and flexibility of a theorem prover with the power of an abstract interpretation-based static analyzer. The verification conditions are generated from the program source, and they are passed to an automatic theorem prover. The prover tries to prove the verification conditions, and it invokes dynamically the abstract interpreter for the inference of (more precise) loop invariants on a *subset* of the program traces. The abstract interpreter infers a loop invariant that is particular to this set of execution traces, and passes it back to the theorem

prover, which then continues the proof. We obtain a program analysis that is a form of trace partitioning. The partitioning is value-based (the partition is done on the values of program variables) and automatic (the theorem prover chooses the partitions automatically). We have a protoype implementation of our technique, built as an extension of the Zap theorem prover.

We plan to extend our work in several directions. For the implementation, we plan (i) to perform several optimizations in the interaction of the prover and the abstract interpreter, *e.g.*, by caching the calls to the GenerateFacts routine or by a smarter handling of the analysis of nested loops; and (ii) to include our work in the static program verifier (codenamed Boogie) that is part of the Spec# programming system [3]. In particular, this second point will require some extensions to the theoretical framework presented in this paper: The starting point in Boogie is a language with basic blocks and goto statements (as opposed to the structured source language we have presented here), and we also want to handle recursive procedures (in this case the abstract interpreter will be invoked not only to generate loop invariants, but also procedure summaries, which we are hopeful can be handled in a similar way on demand). It will also be of interest to instantiate the static analyzer with non-numeric abstract domains, as for instance, a domain for shape analysis (*e.g.*, [30]). In this case, we expect the abstract interpreter to infer invariants on the shape of the heap that are useful to prove heap properties such as that a method correctly manipulates a list (*e.g.*, "if the input is an acyclic list, then the output is also an acyclic list"), and that combined with the inference of class invariants [27, 6] may allow the verification of class-level specifications (*e.g.*, "the class implements an acyclic list").

References

0. Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *15th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'88)*, pages 1–11. ACM, January 1988.
1. Thomas Ball, Byron Cook, Shuvendu K. Lahiri, and Lintao Zhang. Zapato: Automatic theorem proving for predicate abstraction refinement. In *Computer Aided Verification, 16th International Conference, CAV 2004*, volume 3114 of *Lecture Notes in Computer Science*, pages 457–461. Springer, July 2004.
2. Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Model Checking Software, 8th International SPIN Workshop*, volume 2057 of *Lecture Notes in Computer Science*, pages 103–122. Springer, May 2001.
3. Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *CASSIS 2004, Construction and Analysis of Safe, Secure and Interoperable Smart devices*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer, 2005.
4. Clark Barrett and Sergey Berezin. CVC Lite: A new implementation of the cooperating validity checker. In *Proceedings of the 16th International Conference on Computer Aided Verification (CAV)*, Lecture Notes in Computer Science. Springer, July 2004.
5. François Bourdoncle. Abstract interpretation by dynamic partitioning. *Journal of Functional Programming*, 2(4):407–423, 1992.
6. Bor-Yuh Evan Chang and K. Rustan M. Leino. Inferring object invariants. In *Proceedings of the First International Workshop on Abstract Interpretation of Object-Oriented Languages*

(AIOOL 2005), volume 131 of *Electronic Notes in Theoretical Computer Science*. Elsevier, January 2005.

7. Patrick Cousot. The calculational design of a generic abstract interpreter. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999.
8. Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages (POPL '77)*, pages 238–252. ACM, January 1977.
9. Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *6th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '79)*, pages 269–282. ACM, 1979.
10. Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *5th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '78)*, pages 84–97. ACM, 1978.
11. Leonardo de Moura and Harald Rueß. Lemmas on demand for satisfiability solvers. In *Fifth International Symposium on the Theory and Applications of Satisfiability Testing (SAT'02)*, May 2002.
12. Cormac Flanagan, Rajeev Joshi, Xinming Ou, and James B. Saxe. Theorem proving using lazy proof explication. In *15th Computer-Aided Verification conference (CAV)*, July 2003.
13. Cormac Flanagan, Rajeev Joshi, and James B. Saxe. An explicating theorem prover for quantified formulas. Technical Report HPL-2004-199, HP Labs, 2004.
14. Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: generating compact verification conditions. In *28th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'01)*, pages 193–205, 2001.
15. Robert W. Floyd. Assigning meanings to programs. In *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposium in Applied Mathematics*, pages 19–32. American Mathematical Society, 1967.
16. Roberto Giacobazzi and Francesco Ranzato. The reduced relative power operation on abstract domains. *Theoretical Computer Science*, 216(1-2):159–211, March 1999.
17. Susanne Graf and Hassen Saïdi. Construction of abstract state graphs via PVS. In *Computer Aided Verification, 9th International Conference (CAV '97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 1997.
18. Maria Handjieva and Stanislav Tzolovski. Refining static analyses by trace-based partitioning using control flow. In *Static Analysis Symposium (SAS '98)*, volume 1503 of *Lecture Notes in Computer Science*, pages 200–215. Springer, 1998.
19. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In *Proceedings of POPL 2004*, pages 232–244. ACM, 2004.
20. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *29th Syposium on Principles of Programming Languages (POPL'02)*, pages 58–70. ACM, 2002.
21. C. A. R. Hoare. An axiomatic approach to computer programming. *Communications of the ACM*, 12:576–580,583, 1969.
22. Bertrand Jeannet, Nicolas Halbwachs, and Pascal Raymond. Dynamic partitioning in analyses of numerical properties. In *Static Analysis Symposium (SAS'99)*, volume 1694 of *Lecture Notes in Computer Science*. Springer, September 1999.
23. K. Rustan M. Leino. Efficient weakest preconditions. *Information Processing Letters*, 93(6):281–288, March 2005.
24. K. Rustan M. Leino, Madan Musuvathi, and Xinming Ou. A two-tier technique for supporting quantifiers in a lazily proof-explicating theorem prover. In *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005*, volume 3440 of *Lecture Notes in Computer Science*, pages 334–348. Springer, April 2005.

25. K. Rustan M. Leino, James B. Saxe, and Raymie Stata. Checking Java programs via guarded commands. In *Formal Techniques for Java Programs*, Technical Report 251. Fernuniversität Hagen, May 1999. Also available as Technical Note 1999-002, Compaq Systems Research Center.
26. Francesco Logozzo. Approximating module semantics with constraints. In *Proceedings of the 19th ACM SIGAPP Symposium on Applied Computing (SAC 2004)*, pages 1490–1495. ACM, March 2004.
27. Francesco Logozzo. *Modular Static Analysis of Object-oriented Languages*. PhD thesis, École Polytechnique, 2004.
28. Laurent Mauborgne and Xavier Rival. Trace partitioning in abstract interpretation based static analyzers. In *14th European Symposium on Programming (ESOP 2005)*, volume 3444 of *Lecture Notes in Computer Science*, pages 5–20. Springer, 2005.
29. Antoine Miné. The octagon abstract domain. In *AST 2001 in WCRE 2001*, IEEE, pages 310–319. IEEE CS Press, October 2001.
30. Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, 2002.
31. Greta Yorsh, Thomas W. Reps, and Shmuel Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, pages 530–545, 2004.
32. Karen Zee, Patrick Lam, Viktor Kuncak, and Martin C. Rinard. Combining theorem proving with static analysis for data structure consistency. In *International Workshop on Software Verification and Validation*, November 2004.