Assertion Checking over Combined Abstraction of Linear Arithmetic and Uninterpreted Functions

Sumit Gulwani[1] and Ashish Tiwari[2]

[1] Microsoft Research, Redmond, WA 98052
sumitg@microsoft.com
[2] SRI International, Menlo Park, CA 94025
tiwari@csl.sri.com

Abstract. This paper presents results on the problem of checking equality assertions in programs whose expressions have been abstracted using combination of linear arithmetic and uninterpreted functions, and whose conditionals are treated as non-deterministic.

We first show that the problem of assertion checking for this combined abstraction is coNP-hard, even for loop-free programs. This result is quite surprising since assertion checking for the individual abstractions of linear arithmetic and uninterpreted functions can be performed efficiently in polynomial time.

Next, we give an assertion checking algorithm for this combined abstraction, thereby proving decidability of this problem despite the underlying lattice having infinite height. Our algorithm is based on an important connection between unification theory and program analysis. Specifically, we show that weakest preconditions can be strengthened by replacing equalities by their unifiers, without losing any precision, during backward analysis of programs.

1 Introduction

We use the term *equality assertion* or simply *assertion* to refer to an equality between two program expressions. By *assertion checking*, we mean checking whether a given assertion is an invariant at a given program point.

Reasoning about assertions in programs is an undecidable problem. Hence, assertion checking is typically performed over some (sound) abstraction of the program. This may give rise to false positives, i.e., some assertions that are true in the original program may not be true in the abstract version. There is an efficiency-precision trade-off in the choice of the abstraction. A more precise abstraction leads to fewer false positives but is also harder to reason about.

Linear arithmetic and uninterpreted functions[1] are two most commonly used expression languages for creating program abstractions. There are several

[1] An uninterpreted function F of arity n satisfies only one axiom: If $e_i = e'_i$ for $1 \leq i \leq n$, then $F(e_1, .., e_n) = F(e'_1, .., e'_n)$. Uninterpreted functions are commonly used to abstract programming language operators that are otherwise hard to reason about. They are also used to abstract procedure calls.

P. Sestoft (Ed.): ESOP 2006, LNCS 3924, pp. 279–293, 2006.

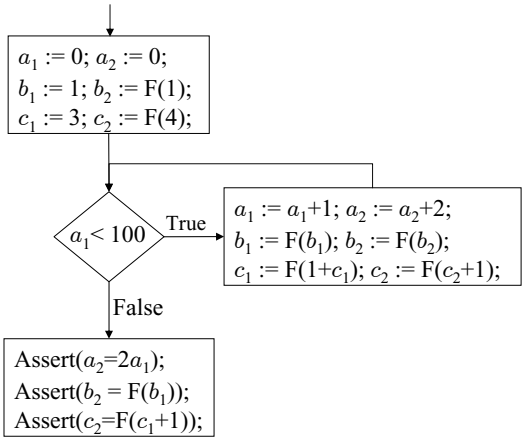

Fig. 1. This program illustrates the difference between precision of performing analysis over the abstractions of linear arithmetic (which can verify only the first assertion), uninterpreted functions (which can verify only the second assertion), and their combination (which can verify all assertions). F denotes some function without any side-effects and can be modeled as an uninterpreted function for purpose of proving the assertions.

papers that describe how to do assertion checking for each of these abstractions. (Section 6 on related work describes some of this work.) The combined expression language of linear arithmetic and uninterpreted functions yields a more precise abstraction than the ones obtained from either of these two expression languages. For example, consider the program shown in Figure 1. Note that all assertions at the end of the program are true. If this program is analyzed over the abstraction of linear arithmetic (using, for example, the abstract interpreter described in [14] or [6]), then only the first assertion can be validated. This is because discovering the relationship between b_1 and b_2, and between c_1 and c_2, involves reasoning about uninterpreted functions. Similarly, if this program is analyzed over the abstraction of uninterpreted functions (using, for example, the abstract interpreter described in [10]), then only the second assertion can be validated. However, an analysis over the combined abstraction of linear arithmetic and uninterpreted functions can verify all assertions.

Even though there has been a lot of work for reasoning about the abstractions of linear arithmetic and that of uninterpreted functions, the problem of assertion checking over the combined abstraction of linear arithmetic and uninterpreted functions has not been considered before. In this paper, we consider the problem of checking equality assertions in programs whose expressions have been abstracted using linear arithmetic and uninterpreted functions. We also abstract all program conditionals as non-deterministic because otherwise the problem is easily shown to be undecidable even for the individual abstractions of linear arithmetic [17] and uninterpreted functions [16]. (An analysis that performs an imprecise reasoning over the combined abstraction of linear arithmetic and uninterpreted functions but takes conditional guards into account would

also be useful in practice, and can be used, for example, for array bounds checking. The related work section mentions our recent work on combining abstract interpreters, which can be used to construct such an analysis.) The abstracted program model is formally described in Section 2.

In Section 3, we show that the problem of assertion checking in the combined abstraction of linear arithmetic and uninterpreted functions is coNP-hard. This is true even for loop-free programs, in which case it is coNP-complete. This result is quite surprising because assertion checking in the individual abstractions of linear arithmetic and uninterpreted functions entails polynomial-time algorithms (even for programs with loops). Karr's algorithm [14, 17] can be used to perform assertion checking when program expressions have been abstracted using linear arithmetic operators. Gulwani and Necula's algorithm [9, 10] performs assertion checking in programs whose expressions have been abstracted using uninterpreted functions. Both these algorithms run in polynomial time. However, our coNP-hardness result shows that there is no way to combine these algorithms to do assertion checking for the combined abstraction in polynomial time (unless P=coNP). A similar combination problem has been studied extensively in the context of decision procedures. Nelson and Oppen have given a famous combination result for combining decision procedures for disjoint, convex and quantifier-free theories with only polynomial-time overhead [20]. The theories of linear arithmetic and uninterpreted functions are disjoint, convex, and quantifier-free and have polynomial time decision procedures. Hence, the Nelson-Oppen combination methodology can be used to construct a polynomial-time decision procedure for the combination of these theories. In this paper, we show that, unfortunately, there is no polynomial-time combination scheme for assertion checking in the combined abstraction of linear arithmetic and uninterpreted functions (unless P=coNP).

In Section 4, we give an assertion checking algorithm for the combined abstraction (of linear arithmetic and uninterpreted functions) thereby showing that this problem is decidable. This result is again surprising because the underlying abstract lattice has infinite height, which implies that a standard abstract interpretation [6] based algorithm cannot terminate in a finite number of steps. However, our algorithm leverages the fact that our goal is not to discover all equality invariants, but to check whether a given assertion is an invariant. A central component of our algorithm is a *general* result that allows replacement of equalities generated in weakest precondition computation by their unifiers (Lemma 2). For theories that admit a singleton or finite complete set of unifiers, respectively called unitary and finitary theories, this replacement can be effectively done. The significance of this connection between assertion checking and unification is discussed further in Section 5. We make the paper self-contained by presenting (in Section 4.1) a novel unification algorithm for the combined theory of linear arithmetic and uninterpreted functions, which is used in our assertion checking algorithm.

2 Program Model

We assume that each procedure in a program is abstracted using the flowchart nodes shown in Figure 2. In the assignment node, x refers to a program variable while e denotes some expression in the underlying abstraction. We refer to the language of such expressions as *expression language of the program*. The expression languages for the abstractions of linear arithmetic, uninterpreted functions and their combination are as follows:

- Linear arithmetic:

$$e ::= y \mid c \mid e_1 \pm e_2 \mid c \times e$$

  Here y denotes some variable while c denotes some arithmetic constant.
- Uninterpreted functions:

$$e ::= y \mid F^n(e_1, .., e_n)$$

  Here F^n denotes some uninterpreted function of arity n. We allow n to be zero (for representing nullary uninterpreted functions).
- Combination of linear arithmetic and uninterpreted functions:

$$e ::= y \mid c \mid e_1 \pm e_2 \mid c \times e \mid F^n(e_1, .., e_n)$$

A non-deterministic assignment $x :=?$ denotes that the variable x can be assigned any value. Such non-deterministic assignments are used as a safe abstraction of statements (in the original source program) that our abstraction cannot handle precisely.

Non-deterministic conditionals, represented by $*$, denote that the control can flow to either branch irrespective of the program state before the conditional. They are used as a safe abstraction of guarded conditionals, which our abstraction cannot handle precisely. We abstract away the guards in conditionals because otherwise the problem of assertion checking (when the expression language of the program involves combination of linear arithmetic and uninterpreted functions) can be easily shown undecidable from either of the following two results. Müller-Olm and Seidl have shown that the problem of assertion

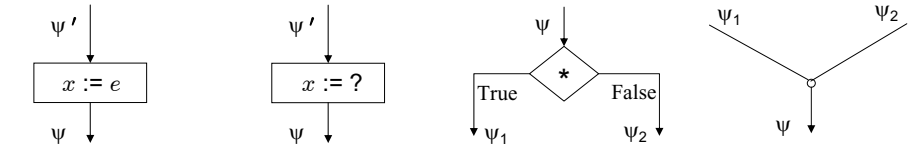

Fig. 2. Flowchart nodes in our abstracted program model

checking in programs that use guarded conditionals and linear arithmetic expressions is undecidable [17]. Müller-Olm, Rüthing, and Seidl have also proved a similar undecidability result when the expression language involves uninterpreted functions [16].

A join node has two incoming edges. Note that a join node with more than two incoming edges can be reduced to multiple join nodes each with two incoming edges.

3 coNP-Hardness of Assertion Checking

In this section, we show that the problem of assertion checking when the expression language of the program involves combination of linear arithmetic and uninterpreted functions (and the flowchart representation of the program consists of nodes shown in Figure 2) is coNP-hard.

The key observation in proving this result is that a disjunctive assertion of the form $g = a \vee g = b$ can be encoded as the non-disjunctive assertion $F(a) + F(b) = F(g) + F(a + b - g)$. The procedure `Check`(g,m) generalizes this encoding for the disjunctive assertion $g = 0 \vee .. \vee g = m - 1$ (which has $m - 1$ disjuncts), as stated in Lemma 1. Once such a disjunction can be encoded, we can reduce the unsatisfiability problem to the problem of assertion checking as follows.

Consider the program shown in Figure 3. We will show that the assert statement in the program is true iff the input boolean formula ψ is unsatisfiable. Note that, *for a fixed* ψ, the procedures `IsUnSatisfiable` and `Check` can be reduced to one procedure whose flowchart representation consists of only the nodes shown in Figure 2. (These procedures use procedure calls and loops with guarded conditionals only for expository purposes.) This can be done by unrolling the loops and inlining procedure `Check` inside procedure `IsUnSatisfiable`. The size of the resulting procedure is polynomial in the size of the input boolean formula ψ.

The procedure `IsUnSatisfiable` contains k non-deterministic conditionals, which together choose a truth value assignment for the k boolean variables in the input boolean formula ψ, and accordingly set its clauses to true (1) or false (0). The boolean formula ψ is unsatisfiable iff at least one of its clauses remains unsatisfied in every truth value assignment to its variables, or equivalently, $g \in \{0, .., m-1\}$ in all executions of the procedure `IsUnSatisfiable`. The procedure `Check`(g, m) performs the desired check as stated in the following lemma.

Lemma 1. *The assert statement in* `Check`(g, m) *is true iff* $g \in \{0, .., m - 1\}$.

Proof. The following properties hold for all $0 \leq i \leq m - 1$.

E1. If $0 \leq j \leq i$, then $h_{i,j} = h_{i,0}$.
E2. If $g \in \{0, .., m - 1\}$, then $h_i = h_{i,g}$.
E3. If $g \notin \{0, .., m - 1\}$, then h_i cannot be expressed as a linear combination of $h_{i,0}, .., h_{i,m-1}$.

```
IsUnSatisfiable(ψ)
   % Suppose formula ψ has k variables x_1,..,x_k
   %             and m clauses numbered 1 to m.
   % Let variable x_i occur in positive form in clauses # A_i[0],..,A_i[c_i]
   %                  and in negative form in clauses # B_i[0],..,B_i[d_i].
   for i = 1 to m do
       e_i := 0; % e_i represents whether clause i is satisfiable or not.
   for i = 1 to k do
       if (*) then % set x_i to true
           for j = 0 to c_i do
               e_{A_i[j]} := 1;
       else % set x_i to false
          for j = 0 to d_i do
               e_{B_i[j]} := 1;
   g := e_1 + e_2 + .. + e_m; % Count how many clauses have been satisfied.
   Check(g, m);

Check(g, m)
   % This procedure checks whether g ∈ {0,..,m-1}.
   h_0 := F(g);
   for j = 0 to m-1 do
       h_{0,j} := F(j);
   for i = 1 to m-1 do
       s_{i-1} := h_{i-1,0} + h_{i-1,i};
       h_i := F(h_{i-1}) + F(s_{i-1} - h_{i-1});
       for j = 0 to m-1 do
           h_{i,j} := F(h_{i-1,j}) + F(s_{i-1} - h_{i-1,j});
   Assert(h_{m-1} = h_{m-1,0});
```

Fig. 3. A program that illustrates the coNP-hardness of assertion checking when the expression language uses combination of linear arithmetic and uninterpreted functions.

The above properties can be proved easily by induction on i. If $g \in \{0,..,m-1\}$, then the assert statement is true because:

$$\begin{aligned} h_{m-1} &= h_{m-1,g} \text{ (follows from property E2)} \\ &= h_{m-1,0} \text{ (follows from property E1)} \end{aligned}$$

If $g \notin \{0,..,m-1\}$, then it follows from property E3 that the assert statement is falsified. ■

Lemma 1 implies that the assert statement in procedure `IsUnSatisfiable`(ψ) is true iff the input boolean formula ψ is unsatisfiable. Hence, the following theorem holds.

Theorem 1. *Assertion checking for programs with non-deterministic conditionals and whose expression language is a combination of linear arithmetic and uninterpreted functions is coNP-hard.*

Since `IsUnSatisfiable` can be represented as a loop-free program, Theorem 1 holds even for loop-free programs.

4 Algorithm for Assertion Checking

In this section, we give an assertion checking algorithm for our abstracted program model when the expression language of the program involves combination of linear arithmetic and uninterpreted functions. We prove that this algorithm terminates, which establishes the decidability of assertion checking for the combined abstraction. It remains an open problem to establish an upper complexity bound for this algorithm.

For purpose of describing and proving correctness of our algorithm, we first establish some results on unification in the combined theory of linear arithmetic and uninterpreted functions in the next sub-section.

4.1 Unification in the Combined Theory

A *substitution* σ is a mapping that maps variables to expressions such that for every variable x, the expression $\sigma(x)$ contains variables only from the set $\{y \mid \sigma(y) = y\}$. A substitution mapping σ can be (homomorphically) lifted to expressions such that for every expression e, we define $\sigma(e)$ to be the expression obtained from e by replacing every variable x by its mapping $\sigma(x)$. Often, we denote the application of a substitution σ to an expression e using postfix notation as $e\sigma$. We sometimes treat a substitution mapping σ as the following formula, which is a conjunction of non-trivial equalities between variables and their mappings:

$$\bigwedge_{x:x \neq x\sigma} x = x\sigma$$

A substitution σ is a *unifier* for an equality $e_1 = e_2$ (in theory T) if $e_1\sigma = e_2\sigma$ (in theory T). A substitution σ is a unifier for a set of equalities E if σ is a unifier for each equality in E. A substitution σ_1 is *more-general* than a substitution σ_2 if there exists a substitution σ such that $x\sigma_2 = (x\sigma_1)\sigma$ for all variables x. [2] A set C of unifiers for E is *complete* when for any unifier σ for E, there exists a unifier $\sigma' \in C$ that is more-general than σ. Theories can be classified based on whether all equalities in that theory have a complete set of unifiers whose cardinality is at most 1 (unitary theory), or finite (finitary theory), or whether some equality does not have any finite complete set of unifiers (infinitary theory).

In the remaining part of this section, we show that the combined theory of linear arithmetic and uninterpreted functions is finitary. For this purpose, we describe an algorithm that computes a complete set of unifiers for an equality in the combined theory. We describe this algorithm using a set of inference rules (listed in table 1) in the style of [4].

[2] The more-general relation is reflexive, i.e., a substitution is more-general than itself.

Table 1. Inference rules for unification in the combination theory

Unif0:
$$\frac{(E \cup \{e = e\}, \sigma)}{(E, \sigma)}$$

Unif1:
$$\frac{(E \cup \{x = e\}, \sigma)}{(E\sigma', \sigma\sigma')}$$
if x does not occur in e. Here $\sigma' = \{x \mapsto e\}$ and $E\sigma'$ denotes $\{e_1\sigma' = e_2\sigma' \mid (e_1 = e_2) \in E\}$.

Unif2:
$$\frac{(E \cup \{F(e_1, .., e_n) = F(e'_1, .., e'_n) + e\}, \sigma)}{(E \cup \{e_1 = e'_1, .., e_n = e'_n, e = 0\}, \sigma)}$$

Table 1 describes some inference rules that operate on states. A state (E, σ) is a pair consisting of a set E of equalities (between expressions involving combination of linear arithmetic and uninterpreted functions) and a substitution σ. The Unif0 rule removes trivial equalities from E. The Unif1 rule can be applied after selecting some equality from E that can be rewritten in the form $x = e$ such that variable x does not occur in expression e. The Unif2 rule is applied after selecting some equality that can be rewritten in the form $F(e_1, .., e_n) = F(e'_1, .., e'_n) + e$ for some uninterpreted function F and expressions e_i, e'_i and e.

The notation $\{x_1 \mapsto e_1, .., x_k \mapsto e_k\}$ denotes the substitution mapping that maps variable x_i to e_i (for $1 \leq i \leq k$) and all other variables to themselves. We use the notation $(E, \sigma) \vdash (E', \sigma')$ to denote that the state (E', σ') is obtained from (E, σ) by applying some inference rule. Similarly, $(E, \sigma) \vdash^* (E', \sigma')$ denotes that the state (E', σ') can be obtained from the state (E, σ) by applying some sequence of inference rules.

To generate a complete set of unifiers C for an equality $e_1 = e_2$, we start with the state $(\{e_1 = e_2\}, I)$, where I is the identity mapping, and apply the inference rules repeatedly until no more inference rules can be applied. For all derivations that end with some state of the form $(\emptyset, \sigma)$, we put σ in C. Theorem 2 stated below implies that the set C thus obtained is indeed a set of unifiers for the equality $e_1 = e_2$. Theorem 3 implies that this set C of unifiers is complete. The proofs of these theorems are by induction on the length of the derivation and are given in the full version of this paper [13].

Theorem 2 (Soundness). *If $(E, I) \vdash^* (\emptyset, \sigma)$, then σ is a unifier for E.*

Theorem 3 (Completeness). *Suppose σ is a unifier for E. Then there is a derivation $(E, I) \vdash^* (\emptyset, \sigma_0)$ such that σ_0 is a more-general unifier for E than σ.*

The following theorem implies that the set C is finite.

Theorem 4 (Finite Complete Set of Unifiers). *Every derivation $(E, I) \vdash^* (E', \sigma')$ takes a finite number of steps. Consequently, E has a finite complete set of unifiers.*

The proof of Theorem 4 is given in the full version of this paper [13]. The key proof idea is to show that every derivation $(E, I) \vdash^* (E', \sigma')$ takes a finite

number of steps and then use Konig's lemma to bound the total number of derivations.

We next illustrate the application of the inference system.

Example 1. Consider the following derivation of a unifier for the equality $Fx + Fy = Fa + Fb$.

$$\begin{array}{ll} (\{Fx + Fy = Fa + Fb\}, I) & \\ (\{Fx = Fb, y = a\}, I) & \text{Unif2} \\ (\{x = b, y = a\}, I) & \text{Unif2} \\ (x = b, \{y \mapsto a\}) & \text{Unif1} \\ (\emptyset, \{x \mapsto b, y \mapsto a\}) & \text{Unif1} \end{array}$$

Thus $\{x \mapsto b, y \mapsto a\}$ is a unifier for $Fx+Fy = Fa+Fb$. Note that the alternate choice for the first Unif2 application yields another unifier $\{x \mapsto a, y \mapsto b\}$ for the given equality. No other unifier can be generated by applying the inference rules. Hence, these two unifiers constitute a complete set of unifiers for the given equality.

Example 2. As another example, consider generating a complete set of unifiers for the equality $x + Fx + Fy = a + Fa + F(a + 1)$. Since each variable occurs below an uninterpreted symbol, only the Unif2 rule is applicable. There are four choices, either $x = a$, or $x = a + 1$, or $y = a$, or $y = a + 1$. We show a derivation for the second choice below.

$$\begin{array}{ll} (\{x + Fx + Fy = a + Fa + F(a + 1)\}, I) & \\ (\{x + Fy = a + Fa, x = a + 1\}, I) & \text{Unif2} \\ (\{a + Fa - Fy = a + 1\}, \{x \mapsto a + Fa - Fy\}) & \text{Unif1} \\ (\{a = a + 1, a = y\}, \{x \mapsto a + Fa - Fy\}) & \text{Unif2} \\ (\{0 = 1\}, \{x \mapsto a + Fa - Fy, y \mapsto a\}) & \text{Unif1} \end{array}$$

The above derivation is now stuck with no inference rule being applicable. Note that only the first choice $x = a$ and the fourth choice $y = a + 1$ successfully generate a unifier, which in both cases is $\{x \mapsto a, y \mapsto a + 1\}$. This unifier yields a singleton complete set of unifiers for the given equality.

4.2 Algorithm

Our algorithm for assertion checking over the combined abstraction is based on weakest precondition computation. It represents invariants at each program point by a formula that is a disjunction of substitution mappings. We show that any program invariant in our abstracted program model can be represented using such formulas (Lemma 2).

Suppose the goal is to check whether an assertion $e_1 = e_2$ is an invariant at program point π. The algorithm performs a backward analysis of the program computing a formula ψ (which is a disjunction of substitution mappings) at each program point such that ψ must hold for the assertion $e_1 = e_2$ to be true at program point π. This formula is computed at each program point from the

formulas at the successor program points in an iterative manner. The algorithm uses the transfer functions described below to compute these formulas across the flowchart nodes shown in Figure 2. The algorithm declares $e_1 = e_2$ to be an invariant at π if the formula computed at the beginning of the program after fixed-point computation is a tautology in the combined theory of linear arithmetic and uninterpreted functions.

In the following transfer functions, we use the notation $\texttt{Unif}(E)$, where E is some conjunction of equalities E, to denote the formula that is a disjunction of all unifiers in some complete set of unifiers for E. (If E is unsatisfiable, then E does not have any unifier and $\texttt{Unif}(E)$ is simply *false*.) The formula $\texttt{Unif}(E)$ can be computed by using the algorithm described in Section 4.1.

Initialization: The formula at all program points except π is initialized to be the trivial formula *true*. The formula at program point π is initialized to be $\texttt{Unif}(e_1 = e_2)$.

Assignment Node: See Figure 2 (a). The formula ψ' before an assignment node $x := e$ is obtained from the formula ψ after the assignment node by substituting x by e in ψ, and invoking `Unif` on each resulting disjunct.

$$\psi' = \bigvee_i \texttt{Unif}(\psi^i[e/x]), \text{ where } \psi = \bigvee_i \psi^i$$

Non-deterministic Assignment Node: See Figure 2 (b). The formula ψ' before a non-deterministic assignment node $x :=?$ is obtained from the formula ψ after the non-deterministic assignment node by substituting program variable x by some fresh constant (i.e., a fresh nullary uninterpreted function symbol) α, and invoking `Unif` on each resulting disjunct.

$$\psi' = \bigvee_i \texttt{Unif}(\psi^i[\alpha/x]), \text{ where } \psi = \bigvee_i \psi^i$$

Non-deterministic Conditional Node: See Figure 2 (c). The formula ψ before a non-deterministic conditional node is obtained by taking the conjunction of the formulas ψ_1 and ψ_2 on the two branches of the conditional, and invoking `Unif` on each resulting disjunct.

$$\psi = \bigvee_{i,j} \texttt{Unif}(\psi_1^i \wedge \psi_2^j), \text{ where } \psi_1 = \bigvee_i \psi_1^i \text{ and } \psi_2 = \bigvee_j \psi_2^j$$

Join Node: See Figure 2 (d). The formulas ψ_1 and ψ_2 on the two predecessors of a join node are same as the formula ψ after the join node.

$$\psi_1 = \psi \text{ and } \psi_2 = \psi$$

Fixed-Point Computation: In presence of loops in procedures, the algorithm goes around each loop until the formulas computed at each program point in two successive iterations of a loop are equivalent, or if any formula becomes *false*.

Correctness. We now prove that the above algorithm is correct, i.e., an assertion $e_1 = e_2$ holds at program point π iff the algorithm claims so. For this purpose, we first state a useful lemma (Lemma 2) that states an interesting connection between program analysis and unification theory. This lemma is true is general: it is independent of the logical theory and also holds for programs with guarded conditionals. The proof of this lemma is given in the full version of this paper [13].

Lemma 2. *An equality $e_1 = e_2$ holds at a program point π iff* $\texttt{Unif}(e_1 = e_2)$ *holds at π. In fact, a formula ϕ containing $e_1 = e_2$ holds at a program point π iff $\phi[\texttt{Unif}(e_1 = e_2)/(e_1 = e_2)]$ holds at π.*

Lemma 2 implies that the formula computed by our algorithm before the flowchart is the (real) weakest precondition of the formula after those nodes. Also, note that the algorithm starts with a formula which is an invariant at π iff the given assertion is an invariant at π (follows from Lemma 2). The correctness of the algorithm now follows from the fact that the algorithm starts with the correct assertion at π and iteratively computes the correct weakest precondition at each program point in a backward analysis.

Termination. We now prove that the above algorithm terminates in a finite number of steps. It suffices to show that the weakest precondition computation across a loop terminates in a finite number of iterations. This follows from the following lemma.

Lemma 3. *Let C be a chain $\psi_1, \psi_2, \ldots$ of formulas that are disjunctions of substitutions. Let $\psi_i = \bigvee_{\ell=1}^{m_i} \psi_i^\ell$ for some integer m_i and substitutions ψ_i^ℓ. Suppose*

(a) $\psi_{i+1} = \bigvee_{\ell=1}^{m_i} \bigvee_{j=1}^{n_i} \texttt{Unif}(\psi_i^\ell \wedge \alpha_i^j)$*, for some substitutions α_i^j.*

(b) $\psi_i \not\Rightarrow \psi_{i+1}$.

Then, C is finite.

The proof of Lemma 3 is by establishing a well founded ordering on $\psi_i's$, and is given in the full version of this paper [13]. Lemma 3 implies termination of our assertion checking algorithm. (Note that the weakest preconditions $\psi_1, \psi_2, \ldots$ generated by our algorithm at any given program point inside a loop in successive iterations satisfy condition (a), and hence $\psi_{i+1} \Rightarrow \psi_i$ for all i. Lemma 3 implies that there exists j such that $\psi_j \Rightarrow \psi_{j+1}$ and hence $\psi_j \equiv \psi_{j+1}$, at which point the fixed-point computation across that loop terminates.) Hence, the following theorem holds.

Theorem 5. *Assertion checking for programs with non-deterministic conditionals and whose expression language is a combination of linear arithmetic and uninterpreted functions is decidable.*

The decidability of assertion checking for the combined abstraction is rather surprising given that the abstract lattice over sets of equalities between expressions

in the combined theory has an infinite height. This suggests that an abstract interpretation based forward analysis algorithm that operates over this lattice may not terminate across loops (unless widening techniques are employed, which may lead to imprecise analysis). For example, consider the following program.

```
InfiniteHeightExample()
   x := 0;
   while (*) do { x := x + 1 };
   Assert(x = 0 ∨ ··· ∨ x = m);
```

The disjunctive assertion at the end of the program can be encoded using an equality assertion. The procedure `Check`(x,m) (on page 284) does exactly this. Clearly, the assertion at the end of the program is not true. To invalidate this assertion, the abstract interpreter will have to go around the loop m times. Hence, it will not terminate across loops (because if it did terminate in say t steps, then it will not be able to invalidate the assertion $x = 0 \vee \cdots \vee x = t$). Our algorithm terminates because it performs a backward analysis (which is good enough for assertion checking) instead of performing a forward analysis (which is required for discovering all valid equalities).

5 Assertion Checking and Unification

The results in this paper point out an interesting connection between assertion checking in programs over a given abstraction and the unification problem for the theory defining that abstraction. Lemma 2 implies that we can replace an assertion by a formula representing a complete set of unifiers for that assertion. This result is quite general and holds for programs with even guarded conditionals and any expression language. This allows for strengthening of weakest preconditions computed using standard transfer functions, by applying `Unif`() to the result *without* losing any precision. This observation is the basis for the close connection between assertion checking and unification.

The theories of linear arithmetic and uninterpreted functions are unitary. However, equalities in the combined theory of linear arithmetic and uninterpreted functions may not have a complete set of unifiers with a cardinality of at most 1. This disparity appears to be responsible for the coNP-hardness of assertion checking for the combined abstraction of linear arithmetic and uninterpreted functions (as opposed to the fact that the abstractions of linear arithmetic and uninterpreted functions have polynomial-time assertion checking algorithms [14, 10]). The presence of multiple unifiers in a minimal complete set allows for encoding of disjunctions in the combined abstraction. For example, the assertion $F(x) + F(3 - x) = F(1) + F(2)$ has two unifiers $x = 1$ and $x = 2$ in its minimal complete set of unifiers. This assertion will be true at any program point iff $x = 1$ or $x = 2$ on all paths leading to this assertion.

The decidability of assertion checking for the combined abstraction (of linear arithmetic and uninterpreted functions) can be attributed to fact that the combined theory is finitary. Observe that the weakest precondition computation of

an assertion, as described in Section 4.2, terminates across a loop because there are only finitely many ways that the assertion can be true.

6 Related Work

We are not aware of any work related to assertion checking for the combined abstraction of linear arithmetic and uninterpreted functions. However, there has been a lot of work on assertion checking and invariant generation over individual abstractions of linear arithmetic and uninterpreted functions.

Program Analysis over Abstraction of Linear Arithmetic. Karr described an algorithm to reason about programs using the abstraction of linear equalities. This algorithm performs a forward analysis of the program and computes a set of linear equalities at each program point [14, 17] in an iterative manner. Gulwani and Necula gave a randomized algorithm that performs an equally precise reasoning but more efficiently [8]. Cousot gave a more precise algorithm that reasons about programs using the abstraction of linear inequalities wherein the facts computed at each program point are linear inequality relationships between program variables [7]. Müller-Olm and Seidl have described a modular linear arithmetic analysis to reason about finite-bit machine arithmetic [19]. There has also been some work on extending some of these analyses to an interprocedural setting [18, 11].

Program Analysis over Abstraction of Uninterpreted Functions. Kildall's algorithm [15] performs abstract interpretation over the lattice of sets of Herbrand equivalences (i.e., equivalences between expressions involving uninterpreted functions) but it runs in exponential time. Alpern, Wegman, and Zadeck (AWZ) gave a polynomial-time algorithm that reasons about programs treating all operators as uninterpreted functions [1]. The AWZ algorithm is less precise than Kildall's algorithm, but is quite popularly used for global value numbering in compilers. Rüthing, Knoop and Steffen's (RKS) polynomial-time algorithm also reasons about programs using the abstraction of uninterpreted functions. The RKS algorithm is more precise than the AWZ algorithm but remains less precise than Kildall's algorithm. Recently, Gulwani and Necula gave a polynomial-time algorithm that is as precise as Kildall's algorithm with respect to assertion checking in programs using the abstraction of uninterpreted functions [9, 10].

Combination of Abstract Interpreters. We have recently described a general methodology to combine abstract interpreters for two abstractions to construct an abstract interpreter for the combination of those abstractions [12]. This methodology can be used to construct an efficient polynomial-time algorithm that performs analysis over the combined abstraction of linear arithmetic and uninterpreted functions and also takes conditional guards into account. However, this algorithm does not perform the most precise reasoning over the combined abstraction of linear arithmetic and uninterpreted functions. Note that the algorithm that we have described in this paper performs the most precise reasoning over the combined abstraction of linear arithmetic and uninterpreted functions, but it does not take conditional guards into account.

Unification for Combination of Theories. The unification problem for the combined theory of linear arithmetic and uninterpreted functions is a simple variant of the unification problem for abelian groups with additional uninterpreted functions. This latter problem is usually referred to as the *general unification problem* for abelian groups [3]. The first algorithm for generating unifiers for the general unification problem for abelian groups was obtained as a corollary of the general result for *combining* unification algorithms [21] and was later refined [2]. The generic combination unification algorithm involves solving the so-called "unification with constants" and "constant elimination" problems [21], or "unification with linear constant restriction" [2] problem for the individual theories. In this paper, we have presented a different unification algorithm for the combined theory of linear arithmetic and uninterpreted functions. Our presentation of this unification algorithm is using inference rules, which are simple to understand and implement.

Decision Procedures for Combination of Theories. Nelson and Oppen gave a general methodology for combining decision procedures for disjoint, convex and quantifier-free theories with only polynomial-time overhead [20]. Shostak gave an efficient variant of this algorithm for the specific case of solvable theories. Clark, Dill and Levitt have described a decision procedure, based on Shostak's method, for combination of linear arithmetic and uninterpreted functions in presence of boolean connectives [5]. It must be mentioned that the problem of assertion checking in programs over a certain abstraction (and in particular for combination of two abstractions) is harder than developing a decision procedure for that abstraction. This is because even though a decision procedure can be used to verify an assertion along a particular program path, a program can potentially have an infinite number of paths. However, if a program is annotated with appropriate invariants at all join points, then a decision procedure can be easily used to verify those invariants as well as assertions across straight-line program fragments.

7 Conclusion

In this paper, we show that assertion checking in programs whose expressions have been abstracted using linear arithmetic and uninterpreted functions is coNP-hard (even for loop-free programs). We also give an algorithm for assertion checking for this abstraction, thereby proving decidability of this problem. These results are obtained by closely analyzing the expressiveness of a theory and its effect on the assertion checking problem. First, the ability to encode disjunctions is identified to be an important factor in making assertion checking hard. Second, the classification of a theory as unitary, finitary, or infinitary—based on whether it admits a singleton, finite, or infinite complete set of unifiers has bearing on the hardness and tractability of the assertion checking problem. We show that assertions can be replaced by their unifiers for purpose of checking if they are invariant. We believe that these observations will be significant when other similar or more general abstractions are considered for program analysis.

References

1. B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *15th Annual ACM Symposium on POPL*, pages 1–11, 1988.
2. F. Baader and K. Schulz. Unification in the union of disjoint equational theories: Combining decision procedures. In *11th International Conference on Automated Deduction*, volume 607 of *LNAI*, pages 50–65, 1992.
3. F. Baader and W. Snyder. Unification theory. In *Handbook of Automated Reasoning*, volume I, chapter 8, pages 445–532. Elsevier Science, 2001.
4. L. Bachmair, A. Tiwari, and L. Vigneron. Abstract congruence closure. *J. of Automated Reasoning*, 31(2):129–168, 2003.
5. C. W. Barrett, D. L. Dill, and J. R. Levitt. Validity checking for combinations of theories with equality. In *First International Conference on Formal Methods in Computer-Aided Design*, volume 1166 of *LNCS*, pages 187–201, 1996.
6. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th Annual ACM Symposium on POPL*, pages 234–252, 1977.
7. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *5th ACM Symposium on POPL*, pages 84–96, 1978.
8. S. Gulwani and G. C. Necula. Discovering affine equalities using random interpretation. In *30th Annual ACM Symposium on POPL*, Jan. 2003.
9. S. Gulwani and G. C. Necula. Global value numbering using random interpretation. In *31st Annual ACM Symposium on POPL*, Jan. 2004.
10. S. Gulwani and G. C. Necula. A polynomial-time algorithm for global value numbering. In *Static Analysis Symposium*, volume 3148 of *LNCS*, pages 212–227, 2004.
11. S. Gulwani and G. C. Necula. Precise interprocedural analysis using random interpretation. In *32nd Annual ACM Symposium on POPL*, Jan. 2005.
12. S. Gulwani and A. Tiwari. Combining abstract interpreters. *Submitted for publication*, Nov. 2005.
13. S. Gulwani and A. Tiwari. Assertion checking over combined abstraction of linear arithmetic and uninterpreted functions. Technical Report MSR-TR-2006-01, Microsoft Research, Jan. 2006.
14. M. Karr. Affine relationships among variables of a program. In *Acta Informatica*, pages 133–151. Springer, 1976.
15. G. A. Kildall. A unified approach to global program optimization. In *1st ACM Symposium on POPL*, pages 194–206, Oct. 1973.
16. M. Müller-Olm, O. Rüthing, and H. Seidl. Checking herbrand equalities and beyond. In *VMCAI*, volume 3385 of *LNCS*, pages 79–96. Springer, Jan. 2005.
17. M. Müller-Olm and H. Seidl. A note on Karr's algorithm. In *31st International Colloquium on Automata, Languages and Programming*, pages 1016–1028, 2004.
18. M. Müller-Olm and H. Seidl. Precise interprocedural analysis through linear algebra. In *31st ACM Symposium on POPL*, pages 330–341, Jan. 2004.
19. M. Müller-Olm and H. Seidl. Analysis of modular arithmetic. In *European Symposium on Programming*, pages 46–60, 2005.
20. G. Nelson and D. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, Oct. 1979.
21. M. Schmidt-Schauss. Unification in a combination of arbitrary disjoint equational theories. *J. Symbolic Computation*, 8(1-2):51–99, 1989.