Assertion Checking Unified

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. We revisit the connection between equality assertion checking in programs and unification that was recently described in [8]. Using a general formalization of this connection, we establish interesting connections between the complexity of assertion checking in programs and unification theory of the underlying program expressions. In particular, we show that assertion checking is: (a) PTIME for programs with nondeterministic conditionals that use expressions from a strict unitary theory, (b) coNP-hard for programs with nondeterministic conditionals that use expressions from a *bitary* theory, and (c) decidable for programs with disequality guards that use expressions from a convex finitary theory. These results generalize several recently published results and also establish several new results. In essence, they provide new techniques for backward analysis of programs based on novel integration of theorem proving technology in program analysis.

1 Introduction

We use the term *equality assertion*, or simply *assertion*, to refer to an equality between two program expressions. The *assertion checking* problem is to decide whether a given assertion always holds at a given program point. In general, assertion checking is an undecidable problem. Hence, assertion checking is typically performed over some sound abstraction of the program. In this paper, we give algorithms as well as hardness results for the assertion checking over classes of useful program abstractions.

Consider, for example, the program shown in Figure 1. All assertions shown in the program are valid (assuming that all variables are integer variables and that there is no overflow). Observe that to prove the validity of the assertions $a = b$ and $y = 2x$, we need to reason about the multiplication operator. Since full reasoning about the multiplication operator is in general undecidable, we can use some sound abstraction of the multiplication operator. One option is to model the multiplication operator as a binary uninterpreted function.[3] Such a model is sufficient to prove the validity of the assertion $a = b$. In Section 4, we show how to use unification algorithm for uninterpreted functions to obtain a polynomial time algorithm for verifying the validity of such assertions.

Modeling the multiplication operator as an uninterpreted function is not sufficient to prove the validity of the assertion $y = 2x$, which requires reasoning about the commutative nature of the multiplication operator. Hence, if we abstract the multiplication operator as a commutative function, we can prove

[3] An uninterpreted function f of arity n satisfies only one axiom: If $e_i = e_i'$ for $1 \leq i \leq n$, then $f(e_1, \ldots, e_n) = f(e_1', \ldots, e_n')$.

validity of the second assertion (as well as the first assertion). However, this requires us to work with program expressions that involve a combination of linear arithmetic and a commutative operator. In Section 5, we show that in general, assertion checking on programs with such program expressions is coNP-hard. However, the good news is that this problem is still decidable, as we show in Section 6. Also observe that the validity of the assertion $y = 2x$ requires the knowledge of the loop guard $\mathit{flag} \neq w$ inside the loop. Our algorithm in Section 6 can also reason about disequality guards and can hence prove the validity of such assertions.

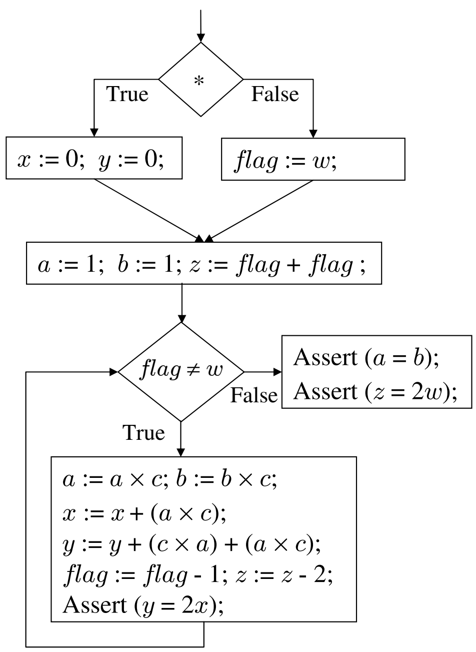

Fig. 1. An example program.

The assertion $z = 2w$ involves discovering the loop invariant $z = 2 \times \mathit{flag}$ and reasoning about the equality guard $\mathit{flag} = w$. Reasoning about such linear arithmetic expressions in the presence of equality guards has been shown to be undecidable in general [13]. This indicates that the decidability results in this paper are *tight* and that one would need incomplete heuristics, such as the one described in Section 7, to reason about arbitrary conditionals. We formalize the notion of reasoning about disequality guards as opposed to reasoning about equality guards by making all conditionals non-deterministic, and introducing `Assume` nodes, as described in Section 2.1.

The main appeal of this work is that all technical results are derived using the basic link between assertion checking for programs whose expressions are from some theory $\mathbb{T}$ and unification in the theory $\mathbb{T}$ (Section 3). An assertion holds at a program point if it evaluates to true in every run of the program. Every run of a program returns a valuation of the program variables. This valuation can be seen as a substitution. If every such substitution makes an assertion *true*, then each substitution would also validate some maximally general $\mathbb{T}$-unifier of the assertion. Using this basic principle, we show that unification algorithms can be used to strengthen assertions during assertion checking using backward analysis. Quite interestingly, the same basic principle also helps us show hardness results in some cases. While this basic principle was presented in an earlier paper [8], its fundamental role in uniformly deriving PTIME , coNP-hardness, and decidability results for assertion checking has been explicated in only this paper.

In particular, the main contributions of this paper are the following general results that relate the *complexity* of assertion checking in programs with

Unification type of theory of program expressions	Complexity of assertion checking	Examples	Generalizes
Strict Unitary	PTIME	ℓa, uf	[7, 12, 13]
Bitary	coNP-hard	ℓa+uf, c	[8]
Finitary-Convex	Decidable	ℓa+uf+ c+ac	[12, 8]

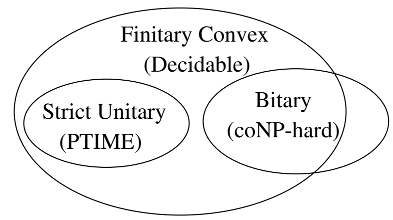

Fig. 2. Summary of results in this paper. If the program model consists of nodes (a)-(d) from Figure 3 and the theory underlying the program expressions belongs to the class given in Col 2, then its assertion checking problem has time complexity given in Col 3. Row 1 requires some additional minor technical assumptions. Row 4 holds even for disequality guards. Col 4 contains examples of theories for which the corresponding result holds:- ℓa: Linear Arithmetic, uf: Uninterpreted Functions, c: Commutative Functions, ac: Associative-Commutative Functions, The symbol + denotes combination of theories. Last column gives references whose results are generalized by our result. The diagram on the right shows containment relationship between theory classes.

the *unification type* of the theory of program expressions. These results are also summarized in Figure 2.

(1) We describe a generic PTIME algorithm for assertion checking in programs when the program expressions are from a strict unitary theory (Section 4).

(2) We introduce the notion of a bitary theory, and prove that several interesting theories (e.g., commutative functions) are bitary. Intuitively, a bitary theory is one that can encode disjunction. We prove that assertion checking in programs whose program expressions are from a bitary theory is coNP-hard (Section 5).

(3) We describe a generic algorithm for assertion checking in programs when the program expressions are from a finitary convex theory, thereby proving decidability. We prove that the (rich) theory of combination of linear arithmetic with functions that are uninterpreted, commutative, or associative-commutative (AC) is finitary and convex (Section 6). The significance of such functions lie in the fact that they can be used to model important properties of otherwise hard to reason about program operators. For example, commutative functions can be used to model floating-point operators (which do not obey associativity), and AC functions can be used to model bit-wise operators.

The above results uniformly generalize several known results [7, 8, 13, 12], and also provide several new results. All prior results on the complexity of assertion checking have been for specific abstractions. For example, in an earlier paper [8] we showed that intraprocedural assertion checking in the combination of linear arithmetic and uninterpreted functions was coNP-hard, but decidable, using a unification based approach. The results in this paper go much beyond one or two specific program abstractions and apply to intra- and inter-procedural analysis of wide classes of program abstractions. They can be used to quickly classify the hardness of these analyses for new abstractions.

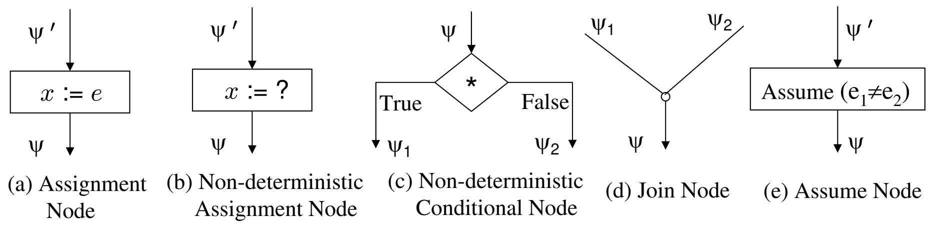

Fig. 3. Flowchart nodes in our abstracted program model.

The results in this paper establish closer connections between program analysis and theorem proving. The traditional way of using theorem proving in program analysis has been via decision procedures. In this usage scenario, decision procedures are used to discharge verification conditions generated from programs annotated with loop invariants. In this paper, theorem proving technology is more tightly integrated in program analysis to make it more precise and efficient, even in the absence of loop-invariant annotations.

The results in this paper should also be viewed in the context of developing new algorithmic techniques for performing *backward* analysis of programs. This paper shows that standard unification algorithms can be used during backward analyses of programs. Finally, although this paper focuses solely on backward analysis, we believe that our observations enable new ways of combining both forward and backward analyses using theorem proving technology to improve overall efficiency and precision [6].

2 Preliminaries

2.1 Program Model

We assume that each procedure in a program is abstracted using the flowchart nodes shown in Figure 3. 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 the *expression language of the program.* Following are examples of the expression languages for some abstractions that we refer to in this paper:

- 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: $e ::= y \mid f(e_1, \ldots, e_n)$
  Here f denotes some uninterpreted function of arity n.
- Commutative Functions $e ::= y \mid f(e_1, e_2)$
  Here f denotes a commutative function.
- Combination of linear arithmetic and uninterpreted functions:
  $e ::= y \mid c \mid e_1 \pm e_2 \mid c \times e \mid f(e_1, \ldots, e_n)$

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.

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

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 can be easily shown to be undecidable even when the program expressions involves operators from simple theories like linear arithmetic [13] or uninterpreted functions [12] (in which case our result in Section 4 would not be possible, and the result in Section 5 would become trivial). This is a very common restriction for a program model while proving preciseness of a program analysis for that model.

However, (for our result in Section 6) we do allow for assume statements of the form $\texttt{Assume}(e_1 \neq e_2)$, which we also refer to as *disequality guards*. Note that a program conditional of the form $e_1 = e_2$ can be reduced to a non-deterministic conditional and assume statements $\texttt{Assume}(e_1 = e_2)$ (on the true side of the conditional) and $\texttt{Assume}(e_1 \neq e_2)$ on the false side of the conditional. Hence, the presence of disequality guards in our program model allows for partial reasoning of program conditionals.

2.2 Unification Terminology

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, i.e., $\bigwedge_x x = x\sigma$.

A substitution σ is a *unifier* for an equality $e_1 = e_2$ (in theory $\mathbb{T}$) if $e_1\sigma = e_2\sigma$ (in theory $\mathbb{T}$). 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. [4] 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. The reader is referred to [1] for an introduction to unification theory.

[4] The more-general relation is reflexive, i.e., a substitution is more-general than itself. All equalities are interpreted modulo theory $\mathbb{T}$.

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*.)

Example 1. Consider the equality $f(x)+f(y) = f(a)+f(b)$ over theory of combination of linear arithmetic and unary uninterpreted function f. The substitution $\{x \mapsto a, y \mapsto b\}$ is a unifier for it. A complete set of unifiers, however, contains two unifiers, viz. $\{x \mapsto a, y \mapsto b\}$ and $\{x \mapsto b, y \mapsto a\}$. Hence,

$$\texttt{Unif}(f(x) + f(y) = f(a) + f(b)) = (x = a \wedge y = b) \vee (x = b \wedge y = a)$$

Theories can be classified based on the cardinality of complete set of unifiers for its equalities as follows.

Unitary Theory A theory $\mathbb{T}$ is said to be *unitary* if for all equalities $e = e'$ in theory $\mathbb{T}$, there exists a complete set of unifiers of cardinality at most 1, that is, there is a unique most-general unifier. We define a unitary theory to be *strict* if for any sequence of equations $e_1 = e_1', e_2 = e_2', \ldots$, the sequence of most-general unifiers $\texttt{Unif}(e_1 = e_1'), \texttt{Unif}(e_1 = e_1' \wedge e_2 = e_2'), \ldots$ contains at most n distinct unifiers where n is the number of variables in the given equations. [5] The theory of linear arithmetic and the theory of uninterpreted functions are both strict unitary.

Bitary Theory We define a theory $\mathbb{T}$ to be *bitary* if there exists an equality $e = e'$ in theory $\mathbb{T}$ such that $y \mapsto z_1$ and $y \mapsto z_2$ form a complete set of unifiers for $e = e'$, where y, z_1 and z_2 are some variables. In other words, $\texttt{Unif}(e = e')$ is $y = z_1 \vee y = z_2$. In addition, we require a technical side condition that for new variables y' and z_1', it is the case that $\texttt{Unif}(e = e[y'/y, z_1'/z_1])$ and $\texttt{Unif}(e' = e'[y'/y, z_1'/z_1])$ are both $y = y' \wedge z_1 = z_1'$.

The theories of a commutative function, combination of linear arithmetic and a unary uninterpreted function, and combination of two associative-commutative functions are all bitary (as proved in Section 5.2). Intuitively, bitary theories are theories that can encode disjunction.

Finitary Theory A theory $\mathbb{T}$ is said to be *finitary* if for all equalities $e = e'$ in theory $\mathbb{T}$, there exists a complete set of unifiers of finite cardinality. Note that every unitary theory is, by definition, finitary. Hence, the theories of linear arithmetic and uninterpreted functions are both finitary. The theory of combination of linear arithmetic and uninterpreted functions is also finitary (as proved in [8]). In this paper, we show that the more general theory of combination of linear arithmetic, uninterpreted functions, commutative functions, and associative-commutative functions is also finitary (Section 6.2).

A theory is said to be *convex* if whenever $e_1 = e_1' \vee e_2 = e_2'$ is valid in the theory, then either $e_1 = e_1'$ is valid in the theory or $e_2 = e_2'$ is valid in the theory. The above-mentioned finitary theories are also convex.

[5] This is an ascending (unifier) chain condition.

```
1 if (*) { x := a; y := b; }
2 else { x := b; y := a; }
3 endif
4 while (*) {
5     x := fx; y := fy;
6     a := fa; b := fb;
7 }
8 assert(x + y = a + b);
```

(a) Program

pc	Assertion at *pc*	
	w/o unification	w/ unification
7	$x + y = a + b$	$x = a + b - y$
4	$x + y = a + b \wedge$ $fx + fy = fa + fb \wedge \cdots$	$(x = a \wedge y = b) \vee$ $(x = b \wedge y = a)$
1	non-termination	*true*

(b) Backward Analysis

1	*true*
3	$(x = a \wedge y = b) \sqcup (x = b \wedge y = a)$ $= (x + y = a + b)$
7	*true*

(c) Forward Analysis

Fig. 4. This figure illustrates the advantage of using unification in backward analysis. The assertion on line 7 of program in Figure (a) is true. Standard backward analysis based procedure, illustrated in Figure (b) Column 1, fails to prove the assertion because it fails to terminate across the loop. Forward analysis in Figure (c) requires *join* computation. Unless we unreasonably assume that the join operator returns the *infinite* set of facts [9], $\bigwedge_i f^i x + f^i y = f^i a + f^i b$, it also fails. When using unification to strengthen assertions in backward analysis, as in Figure (b) Column 2, the fixpoint terminates and we can prove the assertion.

3 Connection between Unification & Assertion Checking

A backward analysis based on weakest precondition computation involves computing assertions at intermediate and initial program points that guarantee that a given assertion holds at a given program point. A unification procedure can be used to strengthen and simplify such assertions. The formula $\texttt{Unif}(E)$ logically implies E, but it is, in general, not equivalent to E. Since it is often "simpler" than E, we may wish to replace $\texttt{assert(E)}$ by $\texttt{assert}(\texttt{Unif}(E))$ at intermediate points during backward analysis. This process is *sound*, that is, if $\texttt{Unif}(E)$ is an invariant, then clearly E will also be an invariant. (See Figure 4 for an example.) But this process is not *complete* in general, that is, if we fail to prove that $\texttt{Unif}(E)$ is an invariant, then we can not conclude anything about E. The basic result formally stated in Lemma 1 and Lemma 2 is that, *in many useful abstractions*, we do *not* lose completeness by this replacement. For instance, unification preserves completeness and helps prove the assertion in the example of Figure 4.

Lemma 1 ([8]). *Let π be any location in a program that is specified using nodes (a)-(d) of Figure 3 and expressions from a theory $\mathbb{T}$. An equality $e = e'$ holds at π iff $\texttt{Unif}_{\mathbb{T}}(e = e')$ holds at π (assuming $\texttt{Unif}_{\mathbb{T}}(e = e')$ is a finite disjunction).*

The proof of this lemma is fairly simple and is given in Appendix A. The key insight is that *runs* of a program are just substitutions and if every run validates an assertion, then every run should also validate some maximally general unifier of that assertion.

We use this soundness and completeness preserving strengthening of assertions in Section 4 as part of a generic PTIME backward analysis procedure for assertion checking in a certain class of programs. Surprisingly, we use this same result to also show *hardness* of assertion checking for another class of programs in Section 5. This simplifies, and simultaneously generalizes, the proof of hardness of assertion checking for a specific theory [8].

We can generalize Lemma 1 as follows to also work in the presence of disequality guards. [6]

Lemma 2. *Let π be a location in a program specified using nodes (a)-(e) of Figure 3 with expressions from a convex finitary theory $\mathbb{T}$. Let ϕ_i be some conjunction of equalities. Then, $\bigvee_i \phi_i$ holds at π iff $\bigvee_i \mathtt{Unif}(\phi_i)$ holds at π.*

The proof of Lemma 2 is given in Appendix B. In Section 6, we argue that the standard backward analysis procedure for assertion checking, *if enhanced by unification based assertion strengthening*, yields a *decision procedure* for a large class of programs.

4 PTIME Decidability for Strict Unitary Theories

In this section, we prove PTIME complexity (by describing a polynomial-time algorithm) for the problem of assertion checking when the expression language of the program comes from a strict unitary theory, and the flowchart representation of the program is abstracted using nodes (a)-(d) shown in Figure 3.

This PTIME complexity result generalizes two earlier known results for theories of linear arithmetic and uninterpreted functions (both of which are unitary theories). Gulwani and Necula gave a polynomial-time algorithm for discovering all assertions of bounded size when the program model consists of nodes (a)-(d) and the expression language consists of uninterpreted functions, thereby proving PTIME complexity of assertion-checking for such programs [7]. Müller-Olm, Rüthing, and Seidl [12] have also pointed out that assertion checking on program with nodes (a)-(d) using the uninterpreted symbols' abstraction (Herbrand equalities) is in PTIME. Muller-Olm and Seidl [13] proved PTIME complexity for assertion checking of programs with nodes (a)-(d) and expression language of linear arithmetic by simplifying Karr's algorithm [11].

4.1 Algorithm

Our algorithm for assertion checking is based on weakest precondition computation. It represents invariants (that need to be satisfied for the assertion to be true) at each program point by a formula that is either *false*, *true*, or a conjunction of equalities of the form $e = e'$.

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

[6] We remark here that the program nodes for which unification does not preserve completeness, viz. *positive guards*, are exactly responsible for *undecidability* of assertion checking for many abstractions.

computing a formula ψ at each program point such that ψ must hold at that program point 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 3. The algorithm declares $e_1 = e_2$ to be an invariant at π iff the formula computed at the beginning of the program after fixed-point computation is *valid*.

Initialization: The formula at all program points except π is initialized to *true*. The formula at program point π is initialized to be $e_1 = e_2$.

Assignment Node: See Figure 3 (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, i.e. $\psi' = \psi[e/x]$.

Non-deterministic Assignment Node: See Figure 3 (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 variable (which does not occur in the program and substitution ψ), i.e. $\psi' = \psi[y/x]$.

Join Node: See Figure 3 (c). 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, i.e. $\psi_1 = \psi$ and $\psi_2 = \psi$.

Non-deterministic Conditional Node: See Figure 3 (d). 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 then pruning away the redundant equations using the `Unif` procedure.

$$\psi = \texttt{UPrune}(\psi_1 \wedge \psi_2)$$

We say an equation $e = e'$ is *redundant* with respect to a formula ψ if $\texttt{Unif}(\psi)$ is a unifier for $e = e'$. The function $\texttt{UPrune}(\psi)$ sequentially checks if each equation $e = e'$ in ψ is redundant with respect to $\psi - \{e = e'\}$ and removes the redundant ones. Thus, $\texttt{Unif}(\psi)$ and $\texttt{Unif}(\texttt{UPrune}(\psi))$ are equivalent.

Fixed-point Computation: In the 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 have *equivalent unifiers*, or if any formula becomes unsatisfiable.

Correctness The correctness of the algorithm follows from the interesting connection between program analysis and unification theory stated in Lemma 1. Specifically, Lemma 1 implies the correctness of pruning and the fixpoint detection steps. It shows that the formula computed by our algorithm before a flowchart node is the weakest precondition of the formula after that node. 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.

Complexity Termination of the fixed-point computation in polynomial time relies on the unitary theory being strict. The following theorem (whose proof is given in Appendix C) states the complexity of the algorithm.

Theorem 1. *Let $\mathbb{T}$ be a strict unitary theory. Suppose that $T_{\mathtt{Unif}}(n)$ is the time complexity for computing the most-general $\mathbb{T}$-unifier of equations given in a shared representation.* [7] *Then the assertion checking problem for programs of size n that are specified using nodes (a)-(d) and whose expressions are from theory $\mathbb{T}$, can be solved in time $O(n^4 T_{\mathtt{Unif}}(n^2))$.*

The above complexity result is conservative because it is based on a generic argument. It can be improved for specific theories, but that is not the focus of this paper.

4.2 Examples of Strict Unitary Theories

If the most-general $\mathbb{T}$-unifiers do not contain any *new variables*, then clearly any chain of increasingly less general substitutions, $\sigma_1, \sigma_1\sigma_2, \sigma_1\sigma_2\sigma_3, \ldots$, will have at most n distinct elements since each new distinct element will necessarily instantiate one uninstantiated variable. This is the case for the theory of linear arithmetic and uninterpreted symbols. The theory of Abelian Groups is unitary, but the most-general unifiers contain new variables. However, using a different argument it can be checked that this theory also satisfies the *strictness* condition.

5 coNP-Hardness for Bitary Theories

In this section, we first show that the problem of assertion checking, when the expression language of the program comes from a bitary theory, is coNP-hard, even when the program is loop-free and the flowchart representation of the program only involves nodes (a)-(d). In the second part of this section, we show that several interesting theories are bitary, thereby establishing that the problem of assertion checking when program expressions are from any of those theories is coNP-hard.

Gulwani and Tiwari [8] showed that the assertion checking problem is coNP-hard when the expression language involves combination of linear arithmetic and uninterpreted functions and when the program model consists of nodes (a)-(d). This section nontrivially generalizes the core idea of the proof of [8], by combining it with the unification connection (Lemma 1), to give a simple characterization of programs for which assertion checking is coNP-hard. This is used to obtain hardness results for several new and unrelated theories.

[7] We assume that the $\mathbb{T}$-unification procedure returns *true* when presented with an equation that is valid (true) in $\mathbb{T}$.

```
Check_T(α_1, ..., α_m, x)
  % Let e = e' be an equality in theory T s.t. Unif(e = e') is y = z_1 ∨ y = z_2.
  e_1 := e[x/y, α_1/z_1, α_2/z_2]; e'_1 := e'[x/y, α_1/z_1, α_2/z_2];
  for j = 1 to m − 2 do
      e_{j+1} := e[e_j/y, e'_j/z_1, e_j[α_{j+2}/x]/z_2]; e'_{j+1} := e'[e_j/y, e'_j/z_1, e_j[α_{j+2}/x]/z_2];
  Assert(e_{m−1} = e'_{m−1});
```

Fig. 5. A procedure that checks whether $(x = \alpha_1) \vee \ldots \vee (x = \alpha_m)$.

5.1 Reduction from 3-SAT

Let $e = e'$ be the equality in theory $\mathbb{T}$ that has $y \mapsto z_1$ and $y \mapsto z_2$ as its complete set of unifiers. The key observation in proving the coNP-hardness result is that a disjunctive assertion of the form $x = \alpha_1 \vee x = \alpha_2$ can be encoded as the non-disjunctive assertion $e_1 = e'_1$, where $e_1 = e[{}^{x}\!/_{y}, {}^{\alpha_1}\!/_{z_1}, {}^{\alpha_2}\!/_{z_2}]$ and $e'_1 = e[{}^{x}\!/_{y}, {}^{\alpha_1}\!/_{z_1}, {}^{\alpha_2}\!/_{z_2}]$. The procedure $\texttt{Check}_{\mathbb{T}}(\alpha_1, \ldots, \alpha_m, x)$ in Figure 5 generalizes this encoding to the disjunctive assertion $x = \alpha_1 \vee \ldots \vee x = \alpha_n$. The unsatisfiability problem can be easily reduced to the problem of checking a disjunctive assertion of the form $x = y_1 \vee \ldots \vee x = y_n$ (where $x, y_1, \ldots, y_n$ are variables). This implies the following theorem (detailed proof in Appendix D).

Theorem 2. *Assertion checking is coNP-hard for (even loop-free) programs specified using nodes (a)-(d) with expressions from the language of a bitary theory.*

5.2 Examples of Bitary Theories

We present a few examples of bitary theories, by presenting a witness equation $e = e'$ for each theory. It is easily verified that $y \mapsto z_1$ and $y \mapsto z_2$ form a complete set of unifiers for $e = e'$ in each theory. Moreover, e and e' can also be verified to satisfy the technical side condition in each case.

The theory of a *commutative function* f can be shown to be bitary using the following equality:

$$f(f(y,y), f(z_1,z_2)) \;=\; f(f(y,z_1), f(y,z_2)) \quad (1)$$

The theory of *combination of linear arithmetic and a unary uninterpreted function* f is also bitary. The following equality is a witness:

$$f(f(y)+f(y)) + f(f(z_1)+f(z_2)) \;=\; f(f(y)+f(z_1)) + f(f(y)+f(z_2)) \quad (2)$$

The theory of *combination of an AC function* g *and a unary uninterpreted function* f is also bitary. The following equality shows this.

$$g(f(g(y,y)), f(g(z_1,z_2))) \;=\; g(f(g(y,z_1)), f(g(y,z_2))) \quad (3)$$

The theory of *combination of two AC functions* f and g is also bitary as shown by the following equality, where c is some constant or a fresh variable distinct from y, z_1 and z_2.

$$g(f(g(y,y),c), f(g(z_1,z_2),c)) \;=\; g(f(g(y,z_1),c), f(g(y,z_2),c)) \quad (4)$$

6 Decidability for Finitary Convex Theories

In this section, we first describe a generic algorithm (thereby proving decidability) for assertion checking when the expression language of the program comes from a finitary theory that is convex, and the flowchart representation of the program consists of nodes (a)–(e) shown in Figure 3. In the second part of this section, we show that the (rich) theory of combination of linear arithmetic, uninterpreted functions, commutative functions, associative-commutative functions is finitary and convex. This establishes the decidability of assertion checking over this theory.

Our result here generalizes, using a uniform framework, the result of Müller-Olm, Rüthing, and Seidl [12] about decidability of checking validity of Herbrand equalities in the presence of disequality guards. It also subsumes our earlier result [8] of decidability of assertion checking for programs whose nodes are restricted to Nodes (a)–(d) and whose expression language involves combination of linear arithmetic and uninterpreted functions. Our new general decidability result is nontrivial since the abstract lattice (underlying the abstractions based on convex finitary theories) often has infinite height, which implies that a standard forward propagation algorithm without widening [3] cannot terminate in a finite number of steps.

6.1 Algorithm

The algorithm is based on weakest precondition computation and is similar to the one described in Section 4. It computes (in a backward analysis) a formula ψ at each program point π such that the formula ψ must hold at π for the given assertion to be true. The formula ψ computed at each program point is either false or a disjunction of conjunction of equalities of the form $x = e$ such that each disjunct represents a valid substitution. Müller-Olm, Rüthing, and Seidl [12] have used a similar representation.

The initialization and the transfer functions for assignment and join nodes are exactly same as the one for the algorithm described in Section 4. We describe the transfer functions for the remaining nodes below.

Non-deterministic Conditional Node: See Figure 3 (d).
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.

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

Assume Node: See Figure 3 (e).
The formula ψ' before an assume node $e_1 \neq e_2$ is obtained from the formula ψ after the assume node as: $\psi' = \psi \vee \texttt{Unif}(e_1 = e_2)$

Correctness and Termination The correctness of the algorithm is an easy consequence of Lemma 2, which shows that unification can be used to strengthen assertions without any loss in soundness or precision. The proof of termination of the algorithm is similar to the proof of termination for the special case of the combined theory of linear arithmetic and uninterpreted functions [8], and is described in Appendix E. Hence, the following theorem holds.

Theorem 3. *Let $\mathbb{T}$ be a convex finitary theory. Then, assertion checking is decidable for programs specified using nodes (a)-(e) with expressions from the language of $\mathbb{T}$.*

6.2 Examples of Finitary Convex Theory

In this section, we prove that the (rich) theory of combination of linear arithmetic, uninterpreted functions, commutative functions, associative-commutative functions is finitary and convex. Let $\mathbb{T}_{LA}, \mathbb{T}_{UF}, \mathbb{T}_C, \mathbb{T}_{AC}$ denote respectively the theories of linear arithmetic, uninterpreted functions, commutative functions, and associative-commutative functions *over disjoint signatures.* Let $\mathbb{T}_{All} = \mathbb{T}_{LA} \cup \mathbb{T}_{UF} \cup \mathbb{T}_C \cup \mathbb{T}_{AC}$. The theory $\mathbb{T}_{All}$ is convex because it is equational. We now use the following well-known result [1] to show that $\mathbb{T}_{All}$ is finitary.

Proposition 1 ([1]). *Let $\mathbb{T}_1, \ldots, \mathbb{T}_n$ be non-trivial equational theories over disjoint signatures that are finitary for $\mathbb{T}_i$-unification with linear constant restrictions. Then $\mathbb{T}_1 \cup \cdots \cup \mathbb{T}_n$ is finitary for elementary unification.*

For a theory $\mathbb{T}$, if unification with constants is finitary, then unification with linear constant restriction, which is more restrictive, is also finitary. Unification with constants is unitary for $\mathbb{T}_{UF}$ and $\mathbb{T}_{LA}$, whereas it is finitary for $\mathbb{T}_C$ and $\mathbb{T}_{AC}$. Therefore, it follows from Proposition 1 that $\mathbb{T}_{All}$ is finitary for elementary unification. Since $\mathbb{T}_{UF}$ is included in $\mathbb{T}_{All}$, it follows that $\mathbb{T}_{All}$ is finitary for general unification as well. In fact, an algorithm to generate the complete set of unifiers in $\mathbb{T}_{All}$ can be obtained using the generic methodology for combining unification algorithms [1].

7 Discussion

Handling Positive Guards. The results in this paper have uniformly assumed that there are no *assume* nodes with positive equalities. In the presence of positive assume nodes, we lose precision if we use unification to replace a weaker assertion by a stronger assertion. This loss in precision is not surprising since the presence of *positive guards* can cause assertion checking to become *undecidable* for several abstractions [13, 12].

In practice, heuristics can be used to deal with positive guards. For instance, the precondition ψ' before a program node `Assume(x=y)` can be obtained from the formula ψ after the assume node as follows: $\psi' \equiv \psi \vee \psi[x/y] \vee \psi[y/x]$. This simple heuristic allows us to prove the assertion $z = 2w$ in the example given in Figure 1. This suggests that the unification based backward analysis procedure proposed in this paper can be effective in practice.

Backward vs. Forward Analysis. Our algorithms for assertion checking are based on backward analysis of programs. Cousot [4] formalized the semantics of sound backward analyses as computing an over-approximation of the set of program states obtained by pushing the negation of the goal backwards, which is equivalent to under-approximation of the set of program states obtained by pushing the goal backwards assuming that the abstract domain is closed under negation. However, abstract domains are, in general, not closed under negation, as is the case for all the equality based abstract domains that we consider in this paper. Additionally, most of these domains do not have precise transfer functions for forward analysis. Hence, there is no automatic recipe to construct algorithms for performing forward or backward analysis of arbitrary abstract domains. This paper shows how to perform precise backward analysis over a large class of abstract domains by using unification algorithms from the corresponding logical theory.

For problems considered in this paper, we can argue that backward analyses are better than forward analyses over corresponding program abstractions in terms of efficiency. This is because performing *precise* assertion checking requires forward analysis to discover *all* facts at each program point, since it is a-priori not clear which facts would be useful to prove the assertion that occurs later in the code. For some of the program abstractions described in this paper (in Section 6), the underlying abstract lattices have infinite height. Hence, forward analyses over those abstractions would not terminate unless widening techniques are used, which would lead to imprecision. However (as surprising as it may be) the backward analyses that we describe in Section 6 terminate over the same abstractions since they only attempts to decide the validity of given assertions (which are finite in number). Figure 4 presents one such example.

On the other hand, forward analyses have their own advantages. The complementary power of forward and backward analyses [4] is not surprising if we realize that these analyses use fundamentally different algorithms to reason over abstractions (that are not closed under negation). For example, forward analyses use quantifier elimination and join/widen algorithms over logical theories as their transfer functions [9]. On the other hand, in this paper, we show how to use unification algorithms to perform backward analysis. These new techniques will, therefore, enable new ways of combining forward and backward analyses in an integrated manner for program verification (e.g., as in [6]).

Connections between Program Analysis and Theorem Proving. This paper contributes to the broader goal of transferring results from the theorem proving community to the world of program analysis [9]. We had earlier shown that forward program analysis can be made more precise and efficient by a tighter coupling with theorem proving technology [9]. In particular, we showed how to use results from Nelson-Oppen combination of decision procedures to generate a more powerful forward analysis by combination of different forward analyses. This paper demonstrates that unification procedures can be useful in improving backward analysis. Unification algorithms have earlier been used in type inferencing [10]. Type inferencing itself can be seen as an abstract interpreter [2]. The results of this paper can be seen as generalizing this basic use of unification in type checking to program analysis over richer abstract domains. Using *backward analysis enhanced with unification*, we showed here that the unification type of a theory determines the complexity of the assertion checking problem for the corresponding abstraction.

8 Conclusion

Unification theory plays a significant role in assertion checking. The unification type of a theory–unitary, bitary, or finitary–is critical in determining the complexity of the assertion checking problem–PTIME, coNP-hard, or decidable–modulo some minor assumptions on the theories and certain restrictions on the program models. These results uniformly generalize several known results and also yield several new ones (see Figure 2). We believe the connections between theorem proving and program analysis developed in this paper can lead to significant new research in both the communities and increase cross-fertilization in unprecedented ways.

References

1. F. Baader and W. Snyder. Unification theory. In *Handbook of Automated Reasoning*, volume I, chapter 8, pages 445–532. Elsevier Science, 2001.
2. P. Cousot. Types as abstract interpretations. In *POPL*, pages 316–331, 1997.
3. 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.
4. P. Cousot and R. Cousot. Refining model checking by abstract interpretation. *Automated Software Engineering*, 6(1):69–95, 1999.
5. N. Dershowitz and Z. Manna. Proving termination with multiset orderings. *Communications of the ACM*, 22(8):465–476, 1979.
6. S. Gulwani and N. Jojic. Program verification as inference in belief networks. Technical Report MSR-TR-2006-98, Microsoft Research, July 2006.
7. 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.
8. S. Gulwani and A. Tiwari. Assertion checking over combined abstraction of linear arithmetic & uninterpreted functions. In *ESOP*, volume 3924 of *LNCS*, Mar. 2006.
9. S. Gulwani and A. Tiwari. Combining abstract interpreters. In *PLDI*, June 2006.
10. R. Hindley. The principal type-scheme of an object in combinatory logic. *Trans. Amer. Math. Soc.*, 146:29–60, 1969.
11. M. Karr. Affine relationships among variables of a program. In *Acta Informatica*, pages 133–151. Springer, 1976.
12. 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.
13. 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.

A Proof of Lemma 1

First we prove an important property of substitutions and the complete set of unifiers.

Lemma 3. *Assuming* $\mathtt{Unif}(\bigwedge_i e_i = e'_i)$ *is a finite disjunction, if* $\mathbb{T} \models \bigwedge_i e_i\sigma = e'_i\sigma$*, then* $\mathbb{T} \models \mathtt{Unif}(\bigwedge_i e_i = e'_i)\sigma$*.*

Proof. Suppose $\mathbb{T} \models \bigwedge_i e_i\sigma = e'_i\sigma$. Let $\mathtt{Unif}(\bigwedge_i e_i = e'_i) = \sigma_1 \vee \cdots \vee \sigma_k$. Since $\{\sigma_1, \ldots, \sigma_k\}$ is a complete set of unifiers of $\bigwedge_i e_i = e'_i$, it follows that there is some j s.t. σ_j is more general than σ, that is, $\sigma =_{\mathbb{T}} \sigma_j\sigma'$ for some σ'. We will show that the formula represented by σ_j becomes $\mathbb{T}$-valid when σ is applied to it. For arbitrary x, consider $x = x\sigma_j$. We need to show that $\mathbb{T} \models x\sigma = x\sigma_j\sigma$. In the theory $\mathbb{T}$, we have

$$\begin{aligned}
x\sigma &=_{\mathbb{T}} x\sigma_j\sigma' && \text{since } \sigma =_{\mathbb{T}} \sigma_j\sigma' \\
&=_{\mathbb{T}} x(\sigma_j\sigma_j)\sigma' && \text{since substitutions are idempotent} \\
&=_{\mathbb{T}} x\sigma_j(\sigma_j\sigma') && \text{since function composition is associative} \\
&=_{\mathbb{T}} x\sigma_j\sigma && \text{since } \sigma =_{\mathbb{T}} \sigma_j\sigma'
\end{aligned}$$

This completes the proof of the lemma. □

We now prove Lemma 1.

Proof. We need to prove that $\mathtt{Unif}(e = e')$ holds at π iff $e = e'$ holds at π. We will show that *in every run* of the program, $\mathtt{Unif}(e = e')$ holds at π iff $e = e'$ holds at π. Therefore, consider an arbitrary run of the program. This will be given by some straight-line programs. Let σ be the substitution that maps each program variable x to the symbolic value of x (in terms of the input variables of the program) at program point π obtained by symbolic execution of the given straight-line program.

We need to show that $\mathbb{T} \models e_1\sigma = e_2\sigma$ iff $\mathbb{T} \models \mathtt{Unif}(e_1 = e_2)\sigma$. The $\Leftarrow$ direction is trivial since $\mathtt{Unif}(e_1 = e_2)$ implies $e_1 = e_2$ (in $\mathbb{T}$). The $\Rightarrow$ direction is a consequence of Lemma 3. □

B Proof of Lemma 2

We first prove a useful lemma.

Lemma 4. *Let* ϕ_i *be a conjunction of equalities for all* i*. If the formula* $\phi_1 \vee \phi_2$ *is valid in a convex theory* $\mathbb{T}$ *then either* ϕ_1 *or* ϕ_2 *is valid in* $\mathbb{T}$*. In general, if the formula* $\bigvee_i \phi_i$ *is valid in* $\mathbb{T}$ *then some* ϕ_i *is valid in* $\mathbb{T}$*.*

Proof. Suppose the claim is false. Let ϕ_1 be $\bigwedge_{i\in I_1} \phi_{1i}$ and ϕ_2 be $\bigwedge_{i\in I_2} \phi_{2i}$, where ϕ_{1i}, ϕ_{2i} are equalities. Since ϕ_1 is not valid in $\mathbb{T}$, there is a $i \in I_1$ such that ϕ_{1i} is not valid in $\mathbb{T}$. Similarly, there is a $j \in I_2$ such that ϕ_{2j} is not valid in $\mathbb{T}$. Therefore, by convexity, the formula $\phi_{1i} \vee \phi_{2j}$ is not valid in $\mathbb{T}$. This means that the formula $\phi_1 \vee \phi_2$ is not valid in $\mathbb{T}$, which contradicts the assumption. The second claim can be proved by generalizing the same argument. □

We are now ready to prove Lemma 2.

Proof. (Lemma 2) $\Rightarrow$: We need to prove that $\bigvee_i \texttt{Unif}(\bigwedge_j e_{ij} = e'_{ij})$ holds at π. In other words, we need to show the formula evaluates to true *in every run* of the program. Therefore, consider an arbitrary run of the program. This will be given by some straight-line code fragment. Let σ be the substitution that maps each program variable x to the symbolic value of x (in terms of the program inputs or the initial values of program variables) at program point π obtained by symbolic execution of the given straight-line program. Let $e_k \neq e'_k$, $k \in K$, be the symbolic evaluations of *all* the assume nodes in the straight-line code. Since $\bigvee_i \bigwedge_j e_{ij} = e'_{ij}$ holds at π, it follows that

$$\begin{array}{rl} & \mathbb{T} \models \bigwedge_{k\in K} e_k \neq e'_k \Rightarrow (\bigvee_i \bigwedge_j e_{ij}\sigma = e'_{ij}\sigma) \\ \text{IFF} & \mathbb{T} \models \bigvee_{k\in K} e_k = e'_k \vee (\bigvee_i \bigwedge_j e_{ij}\sigma = e'_{ij}\sigma) \\ \text{IFF} & \mathbb{T} \models e_k = e'_k \text{ for some } k \in K, \quad \text{OR} \\ & \mathbb{T} \models \bigwedge_j e_{ij}\sigma = e'_{ij}\sigma \text{ for some } i \in I \end{array}$$

The last step is a consequence of Lemma 4. If $\mathbb{T} \models e_k = e'_k$, then $\bigvee_i \texttt{Unif}(\bigwedge_j e_{ij} = e'_{ij})$ holds in this run, and we are done. In the other case, we have $\mathbb{T} \models \bigwedge_j e_{ij}\sigma = e'_{ij}\sigma$, from which it follows using Lemma 3 that $\mathbb{T} \models \texttt{Unif}(\bigwedge_j e_{ij} = e'_{ij})\sigma$.

$\Leftarrow$: This follows from the fact that $\mathbb{T} \models \texttt{Unif}(\bigwedge_j e_{ij} = e'_{ij}) \Rightarrow \bigwedge_j e_{ij} = e'_{ij}$, which is a consequence of the definition of unifiers. $\square$

C Proof of Theorem 1

Proof. Since the program is of size n, the number of variables is bounded by n. Due to the strictness condition, each node in the flowchart changes at most n times. Since there are at most n nodes, there are at most n^2 changes. For each change, we may have to visit all n nodes once. Hence, there are n^3 node visits. In any such visit, `UPrune` is the most complex operation we could perform. In this operation, there are at most $2n$ equations to check for redundancy. The size of each equation, in shared representation, is bounded by n. This is because some path in the program itself contains a representation for the expression in an equation. Thus, pruning takes at most $2n(T_{\texttt{Unif}}(n^2))$ time. Hence the overall time complexity is $O(n^4(T_{\texttt{Unif}}(n^2) + T_{\texttt{Valid}}(n^2)))$. $\square$

D Proof of Theorem 2

Consider the program shown in Figure 6. We will show that the assert statement in the program is true iff the input boolean formula ψ is unsatisfiable. Note that, *for a given* ψ, the procedures `IsUnSatisfiable` and `Check` can be reduced to one procedure whose flowchart representation consists of only the nodes shown in Figure 3. (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 $\texttt{Check}_{\mathbb{T}}$ inside procedure $\texttt{IsUnSatisfiable}_{\mathbb{T}}$. The size of

% Suppose formula ψ has k variables $x_1, \ldots, x_k$ and m clauses numbered 1 to m.
% Let variable x_i occur in positive form in clauses # $A_i[0], \ldots, A_i[c_i]$; and in negative form in clauses # $B_i[0], \ldots, B_i[d_i]$.

```
IsUnSatisfiable_T(ψ)
  % g_i = x_0 represents clause i is unsatisfied
  % g_i = x_1 represents clause i is satisfied.
  for i = 1 to m do
       g_i := x_0;
  for i = 1 to k do
       if (*) then % set x_i to true
            for j = 0 to c_i do
                 g_{A_i[j]} := x_1;
       else % set x_i to false
          for j = 0 to d_i do
                g_{B_i[j]} := x_1;
  % Check if at least one of g_i is unsatisfied.
  Check_T(g_1, ..., g_m, x_0);
```

Fig. 6. A program that illustrates the coNP-hardness of assertion checking when the expression language is from a bitary theory.

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 (x_1) or false (x_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, $\bigvee_{i=1}^{m} g_i = x_0$ in all executions of the procedure `IsUnSatisfiable`.

The procedure `Check`$(g_1, \ldots, g_m, x_0)$ performs the desired check as stated in the following lemma. The key idea in the proof of Lemma 5 is to show that the `Check` procedure constructs an equation whose complete set of unifiers is $\bigvee_{i=1}^{m} x = \alpha_i$. Lemma 5 is then an easy consequence of Lemma 1.

Lemma 5. *The assert statement in* `Check`$(\alpha_1, \ldots, \alpha_m, x)$ *is true iff* $\bigvee_{i=1}^{m} x = \alpha_i$ *holds at the beginning of* `Check`$(\alpha_1, \ldots, \alpha_m, x)$.

Proof. (Lemma 5) We prove by induction on j that `Assert`$(e_j = e'_j)$ holds iff `Assert`$(\bigvee_{i=1}^{j+1} x = \alpha_i)$ holds. Using Lemma 1, it suffices to prove that, if t_j and t'_j are the symbolic terms represented by e_j and e'_j, then `Unif`$(t_j = t'_j)$ is $\bigvee_{i=1}^{j+2} x = \alpha_i$. For $j = 1$, since `Unif`$(e = e')$ is $y = z_1 \vee y = z_2$ by assumption, it follows that `Unif`$(e[{}^{x}\!/\!_{y}, {}^{\alpha_1}\!/\!_{z_1}, {}^{\alpha_2}\!/\!_{z_2}] = e'[{}^{x}\!/\!_{y}, {}^{\alpha_1}\!/\!_{z_1}, {}^{\alpha_2}\!/\!_{z_2}])$ is $x = \alpha_1 \vee x = \alpha_2$ (by variable renaming).

For the induction step, using the same argument, we observe that

$$
\begin{array}{ll}
\quad \texttt{Unif}(e[{}^{e_j}\!/_{y}, {}^{e'_j}\!/_{z_1}, {}^{e_j[\alpha_{j+2}/x]}\!/_{z_2}] = e'[{}^{e_j}\!/_{y}, {}^{e'_j}\!/_{z_1}, {}^{e_j[\alpha_{j+2}/x]}\!/_{z_2}]) & \\
\Leftrightarrow \texttt{Unif}(e_j = e'_j) \vee \texttt{Unif}(e_j = e_j[{}^{\alpha_{j+2}}\!/_{x}]) & \text{(follows from def of bitary theory)} \\
\Leftrightarrow \texttt{Unif}(e_j = e'_j) \vee x = \alpha_{j+1} & \text{(follows from Lemma 6 stated below)} \\
\Leftrightarrow (\bigvee_{i=1}^{j+2} x = \alpha_i) & \text{(follows from Induction hypothesis)} \ \square
\end{array}
$$

Lemma 6. *Let $\mathbb{T}$ be a bitary theory and Let $e = e'$ be the corresponding equation. If $e_1, \ldots, e_{m-1}$ denote the symbolic expressions constructed by program $\texttt{Check}_{\mathbb{T}}$, then $\texttt{Unif}(e_j = e_j[\alpha/x])$ and $\texttt{Unif}(e'_j = e'_j[\alpha/x])$ are both $x = \alpha$.*

Proof. We prove by induction on j. For the base case $j = 1$,

$$
\begin{array}{l}
\quad \texttt{Unif}(e_1 = e_1[\alpha/x]) \\
\Leftrightarrow \texttt{Unif}(e[x/y, \alpha_1/z_1, \alpha_2/z_2] = e[x/y, \alpha_1/z_1, \alpha_2/z_2][\alpha/x]) \\
\quad \text{By definition} \\
\Leftrightarrow x = \alpha \wedge \alpha_1 = \alpha_1 \\
\quad \text{Using the technical side condition in definition of Bitary} \\
\Leftrightarrow x = \alpha
\end{array}
$$

The other case can be obtained by replacing e by e' in the above proof. For the induction step,

$$
\begin{array}{l}
\quad \texttt{Unif}(e_{j+1} = e_{j+1}[\alpha/x]) \\
\Leftrightarrow \texttt{Unif}(e[e_j/y, e'_j/z_1, e_j[\alpha_{j+2}/x]/z_2] = e[e_j/y, e'_j/z_1, e_j[\alpha_{j+2}/x]/z_2][\alpha/x]) \\
\Leftrightarrow \texttt{Unif}(e[y'/y, z'_1/z_1, z'_2/z_2] = e[y''/y, z''_1/z_1, z'_2/z_2] \wedge \\
\qquad\quad y' = e_j \wedge z'_1 = e'_j \wedge z'_2 = e_j[\alpha_{j+2}/x] \wedge \\
\qquad\quad y'' = e_j[\alpha/x] \wedge z''_1 = e'_j[\alpha/x]) \\
\quad \text{By introducing new equational definitions} \\
\Leftrightarrow \texttt{Unif}(y' = y'' \wedge z'_1 = z''_1 \wedge y' = e_j \wedge z'_1 = e'_j \wedge \\
\qquad\quad z'_2 = e_j[\alpha_{j+2}/x] \wedge y'' = e_j[\alpha/x] \wedge z''_1 = e'_j[\alpha/x]) \\
\quad \text{Using the technical side condition in definition of Bitary} \\
\Leftrightarrow \texttt{Unif}(e_j = e_j[\alpha/x] \wedge e'_j = e'_j[\alpha/x]) \\
\quad \text{Removing the dummy variables introduced above} \\
\Leftrightarrow x = \alpha \\
\quad \text{By induction hypothesis}
\end{array}
$$

The other case can be obtained by replacing e by e' in the above proof. $\square$

E Proof of Theorem 3

The correctness of the algorithm in Section 6 is an easy consequence of Lemma 2, which shows that unification can be used to strengthen assertions without any loss in soundness or precision. We now prove that the 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 7. *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) $\psi_{i+1} = \bigvee_{\ell=1}^{m_i} \bigvee_{j=1}^{n_i} \mathtt{Unif}(\psi_i^\ell \wedge \alpha_i^j)$*, for some substitutions* α_i^j.

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

Then, C is finite.

Proof. The key idea behind the proof is to establish a well founded ordering on ψ_i's. We define measure of $\bigvee_{\ell=1}^{m_i} \psi_i^\ell$ to be the multiset $\{k - |\psi_i^\ell| : 1 \leq \ell \leq m_i, \psi_i^\ell \not\equiv \mathit{false}\}$, where k is the total number of variables, and $|\psi_i^\ell|$ denotes the number of conjuncts in ψ_i^ℓ. Since each ψ_i^ℓ is a substitution mapping, this measure is a multiset on natural numbers. We compare two measures using a multiset extension of the ordering $>$ on natural numbers [5].

We now show that the measure of ψ_{i+1} is smaller than that of ψ_i. Since $\psi_i \not\Rightarrow \psi_{i+1}$, there exists $1 \leq \ell \leq m_i$ such that $\psi_i^\ell \not\Rightarrow \alpha_i^j$ for all $1 \leq j \leq n_i$. This implies that for all $1 \leq j \leq n_i$, if $\psi_i^\ell \wedge \alpha_i^j$ is not *false*, then $|\mathtt{Unif}(\psi_i^\ell \wedge \alpha_i^j)| > |\psi_i^\ell|$. Also, note that for all $1 \leq \ell' \leq m_i$ such that $\ell' \neq \ell$, if $\psi_i^{\ell'} \wedge \alpha_i^j$ is not *false*, then $|\mathtt{Unif}(\psi_i^{\ell'} \wedge \alpha_i^j)| \geq |\psi_i^{\ell'}|$ for all $1 \leq j \leq n_i$. Hence, the measure of ψ_{i+1} is smaller than that of ψ_i.

Since the multiset extension of a well-founded ordering is well-founded [5], the measure cannot infinitely decrease. Hence, the chain C is finite. $\square$

Lemma 7 implies termination of the assertion checking algorithm because of the following. 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 7 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.