Assertion Checking Unified *

Sumit Gulwani
Microsoft Research, Redmond, WA, 98052
sumitg@microsoft.com

Ashish Tiwari
SRI International, Menlo Park, CA, 94025
tiwari@csl.sri.com

Abstract

This paper establishes an interesting connection between assertion checking in programs and unification in the theory underlying the program expressions. Using this connection, we describe how unification algorithms from theorem proving can be used to perform backward analysis over programs for assertion checking. Interestingly enough, this connection also helps prove hardness results for assertion checking for classes of program abstractions. In particular, we show
(a) Assertion checking is PTIME for programs with nondeterministic conditionals that use expressions from a strict unitary theory.
(b) Assertion checking is coNP-hard for programs with nondeterministic conditionals that use expressions from a bitary theory.
(c) Assertion checking is decidable for programs with disequality guards that use expressions from a convex finitary theory.
(d) Summary computation for interprocedural analysis can be performed using backward analysis, enabled with unification, on generic assertions. This helps generalize result (a) to interprocedural analysis.
These results generalize several recently published results using a uniform framework. They also provide several new results, and partially solve the long standing open problem of interprocedural global value numbering. 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. 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[1]. 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.

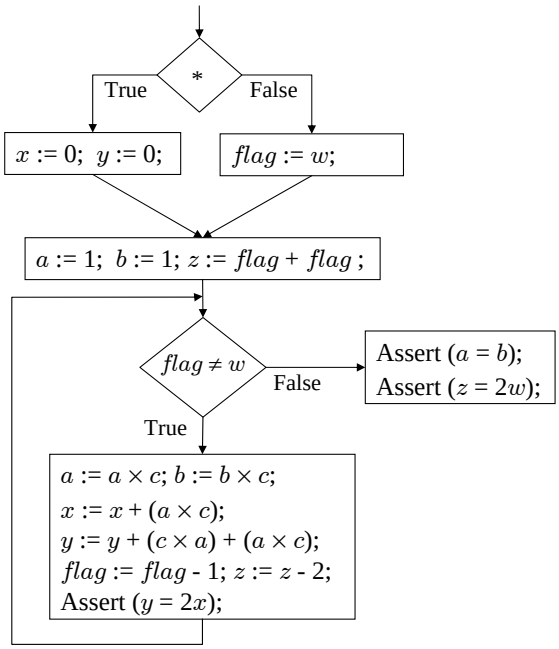


Figure 1. An example program with assertions.

Modeling 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 validity of the second assertion (as well as the first assertion). However, this requires us to work with program expressions that involve 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 $flag \neq w$ inside the loop. Our algorithm in Section 6 can also reason about disequality guards and can hence prove the validity of such assertions.

The assertion $z = 2w$ involves discovering the loop invariant $z = 2 \times flag$ and reasoning about the equality guard $flag = w$. Reasoning about such linear arithmetic expressions in presence of equality guards has been shown to be undecidable in general [15]. This assertion thus points out the limitation of the techniques described in this paper, namely that they cannot reason precisely about the equality guards. However, we do present a heuristic in Section 8 that can also reason about simple examples such as this

* The authors thank Akash Lal and Zhendong Su for providing helpful comments on an earlier version of this paper.

[1] An uninterpreted function f of arity n satisfies only one axiom: If $e_i = e'_i$ for $1 \leq i \leq n$, then $f(e_1, \ldots, e_n) = f(e'_1, \ldots, e'_n)$.

Program nodes	Unification type of theory of program expressions	Assertion checking complexity	Examples	Generalizes
a–d	Strict Unitary	PTIME	LA, UF	[9, 14, 15]
a–d	Bitary	coNP-hard	LA+UF, C	[10]
a–e	Finitary-Convex	Decidable	LA+UF+C+AC	[14, 10]
a–d,f	Strict Unitary	PTIME	LA, Unary	[16]

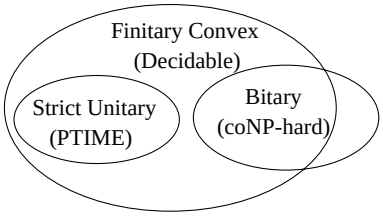


Figure 2. Summary of results in this paper. Program nodes refer to those in Figure 3. If a program contains nodes from Column 1 and the theory underlying the program expressions belongs to the class given in Column 2, then its assertion checking problem has time complexity given in Column 3. Rows 1 and 4 additionally require some additional minor technical assumptions. Column 4 contains examples of theories for which the corresponding result holds:- LA: Linear Arithmetic, UF: Uninterpreted Functions, C: Commutative Functions, AC: Associative-Commutative Functions, Unary: Unary Uninterpreted Functions. The symbol $+$ denotes combination of theories. The last column gives those references whose results are generalized by our result. The diagram on the right shows the containment relationship between the different theory classes.

one. 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 observation at the core of the technical results (described in Section 3) is the connection between assertion checking for programs whose expressions are from some theory $\mathbb{T}$ and unification in the theory $\mathbb{T}$. 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.

In particular, the main contributions of this paper are the following general results that relate the *complexity* of assertion checking in programs with the *unification type* of the theory of program expressions. These results are also summarized in Figure 2.

- We describe a generic PTIME algorithm for assertion checking in programs when the program expressions are from a strict unitary theory (Section 4). We also describe some conditions under which this algorithm can be extended to perform a precise interprocedural analysis in PTIME (Section 7). We show that these conditions are met for the theory of linear arithmetic and unary uninterpreted functions. The latter result partially solves the open problem of interprocedural global value numbering.
- We introduce the notion of a bitary theory, and prove that several interesting theories are bitary. We prove that assertion checking in programs whose program expressions are from a bitary theory is coNP-hard (Section 5). For example, the theory of commutative functions is bitary.
- 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), while AC functions can be used to model bit-wise operators.

The above results uniformly generalize several known results [9, 10, 12, 15, 16, 14], 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 [10] 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. This paper substantially, and nontrivially, generalizes the results of [10]. 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.

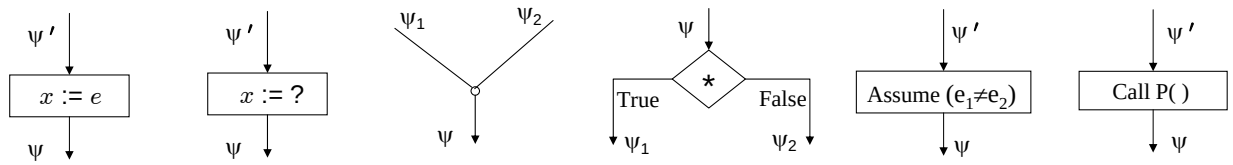
The results in this paper establish closer connections between program analysis and theorem proving. The traditional way of using theorem proving in program analysis has been via decision procedures. In this usage scenario, decision procedures are used to discharge verification conditions generated from 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 intra- and inter-procedural *backward* analysis of programs. This paper shows that standard unification algorithms can be used during intraprocedural backward analyses of programs. This same backward propagation procedure, enabled with unification, can also be used in the interprocedural setting. The only difference is that we need backward propagation and unification to work on generic assertions. This uniformity is appealing from the viewpoint of understanding the difficulty of interprocedural analysis. (Unification on equations arising from generic assertions is often much harder than standard unification.) It also enables, in the specific abstraction of unary uninterpreted symbols, a new polynomial time procedure for interprocedural analysis, which is an important step towards the more general open problem of interprocedural global value numbering problem (when all program operators are treated as uninterpreted). 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 the overall efficiency and precision [7].

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



(a) Assignment Node (b) Non-deterministic Assignment Node (c) Join Node (d) Non-deterministic Conditional Node (e) Assume Node (f) Procedure Call Node

Figure 3. Flowchart nodes in our abstracted program model.

expressions as *expression language of the program*. Following are examples of the expression languages for some abstractions that we refer to in this paper:

- Linear arithmetic:

$$e ::= y \quad | \quad c \quad | \quad e_1 \pm e_2 \quad | \quad c \times e$$

  Here y denotes some variable while c denotes some arithmetic constant.

- Uninterpreted functions:

$$e ::= y \quad | \quad f(e_1, \ldots, e_n)$$

  Here f denotes some uninterpreted function of arity n.

- Combination of linear arithmetic and uninterpreted functions:

$$e ::= y \quad | \quad c \quad | \quad e_1 \pm e_2 \quad | \quad c \times e \quad | \quad f(e_1, \ldots, e_n)$$

- Commutative Functions

$$e ::= y \quad | \quad f(e_1, e_2)$$

  Here f denotes a commutative function.

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 multiple 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 [15] or uninterpreted functions [14] (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.

In Section 7, we show how our techniques can be used to reason about procedure calls. For simplicity, we assume that the inputs and outputs of a procedure are passed as global variables [23]. Hence, the procedure call node simply denotes the name of the procedure to be called.

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:

$$\bigwedge_x x = x\sigma$$

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. [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. The reader is referred to [2] for an introduction to unification theory.

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,*

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

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

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

given equations[3]. 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, 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 [10]). 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 $e_1 = e_1'$ or $e_2 = e_2'$ is valid in the theory. The above-mentioned finitary theories are also convex.

3. Connection between Unification and Assertion Checking

Forward program analysis is based on computing over-approximations of the reachable states. In the process of forward program analysis, over-approximating the states is always *sound*, but not always complete. Backward analysis, in contrast, computes the assertion that must be true at intermediate and initial program points to guarantee that a given assertion holds at a given program point. Under-approximations, that is, replacing an assertion by a stronger assertion, is always *sound* in this case, but not always complete.

Unification procedure can be used to strengthen and simplify an assertion. 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 E by $\texttt{Unif}(E)$ 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 crucial and surprising observation presented in this section 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. *Let π be a program point in a program specified using nodes (a)-(d) of Figure 3 using the expression language of theory $\mathbb{T}$. An equality $e = e'$ holds at a program point π iff $\texttt{Unif}_{\mathbb{T}}(e = e')$ holds at π.*

The proof of this lemma is fairly simple and is given in Appendix **??**. 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.

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

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, our previous result on hardness of assertion checking for a specific theory [10].

We can generalize Lemma 1 to also work in the presence of disequality guards[4] as stated below.

LEMMA 2. *Let π be a program point in a program specified using nodes (a)-(e) of Figure 3 using the expression language of a convex theory $\mathbb{T}$. Let ϕ_i be some conjunction of equalities. Then, $\bigvee_i \phi_i$ holds at a program point π iff $\bigvee_i \texttt{Unif}(\phi_i)$ holds at π.*

The proof is given in Appendix **??**. 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.

This connection between unification and assertion checking is used to develop a novel way to compute *procedure summaries*. We use these summaries for assertion checking in presence of procedure calls (i.e., node (f) in Figure 3). Developed further in Section 7, the main observation is that summary computation involves performing backward analysis, enabled with unification, in the presence of new variables that are required to represent generic assertions at the end of a procedure.

4. PTIME Decidability of Assertion Checking 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 [9]. Müller-Olm, Rüthing, and Seidl [14] 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 [15] proved PTIME complexity for assertion checking of programs with nodes (a)-(d) and expression language of linear arithmetic by simplifying Karr's algorithm [12].

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 computing a formula ψ at each program point such that ψ must hold at that program point for the assertion

[4] 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.

```
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

pc	Assertion
1	*true*
3	$\{x = a \wedge y = b\} \sqcup$ $\{x = b \wedge y = a\}$ $= x + y = a + b$
7	*true*

(c) Forward Analysis

Figure 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 [11], $\bigwedge_i f^i x + f^i y = f^i a + f^i b$, 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.

$e_1 = e_2$ 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.

$$\psi' \quad = \quad \psi[e/x]$$

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).

$$\psi' \quad = \quad \psi[y/x]$$

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.

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

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 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.

4.1.1 Correctness

We now prove that the above algorithm is correct, i.e., an assertion $e = e'$ holds at program point π iff the algorithm claims so. 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.

4.1.2 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 B) states the complexity of the algorithm.

THEOREM 1. *Let $\mathbb{T}$ be a strict unitary theory. Suppose that $T_{\texttt{Unif}}(n)$ is the time complexity for computing the most-general $\mathbb{T}$-unifier of equations given in a shared representation.*[5] *Then the assertion checking problem for programs of size n, which are specified using nodes (a)-(d) and the language of $\mathbb{T}$, can be solved in time $O(n^4 T_{\texttt{Unif}}(n^2))$.*

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 of Assertion Checking 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

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

% 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);
```

```
Check_T(α_1, ..., α_m, x)
  % This procedure checks if (x = α_1) ∨ ... ∨ (x = α_m).
  % 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});
```

Figure 5. A program that illustrates the coNP-hardness of assertion checking when the expression language is from a bitary theory.

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 [10] 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 [10] 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.

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 for the disjunctive assertion $x = \alpha_1 \vee \ldots \vee x = \alpha_m$. 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 5. 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 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 $\texttt{Check}(g_1, \ldots, g_m, x_0)$ performs the desired check as stated in the following lemma.

LEMMA 3. *The assert statement in* $\texttt{Check}(\alpha_1, \ldots, \alpha_m, x)$ *is true iff* $\bigvee_{i=1}^{m} x = \alpha_i$ *holds at the beginning of* $\texttt{Check}(\alpha_1, \ldots, \alpha_m, x)$.

The procedure `Check` constructs an equation whose complete set of unifiers is $\bigvee_{i=1}^{m} x = \alpha_i$. Lemma 3 is then an easy consequence of Lemma 1.

Hence, the following theorem holds.

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:

$$f(f(y,y), f(z_1,z_2)) = f(f(y,z_1), f(y,z_2)) \tag{1}$$

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

$$\begin{aligned} & f(f(y) + f(y)) + f(f(z_1) + f(z_2)) \\ = \;& f(f(y) + f(z_1)) + f(f(y) + f(z_2)) \end{aligned} \tag{2}$$

Note that the equations $f(y) + f(z_1 + z_2 - y) = f(z_1) + f(z_2)$ and $f(f(y)) + f(f(z_1) + f(z_2) - f(y)) = f(f(z_1)) + f(f(z_2))$ also have the same set of unifiers, but they do not satisfy the technical side condition. We can use the latter equation as $e = e'$ in `Check` function and prove coNP-hardness. However the generic hardness proof currently uses a stronger side condition which is only satisfied by Equation 2.

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

$$\begin{aligned} & g(f(g(y,y)), f(g(z_1,z_2))) \\ = \;& g(f(g(y,z_1)), f(g(y,z_2))) \end{aligned} \tag{3}$$

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.

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

Note the similarity between the equalities used in all of the above examples. They are obtained by encoding a commutative function in different theories in different ways.

We conjecture that the hardness result also holds if we drop the technical side condition in the definition of a bitary theory. The technical side condition only identifies a smaller class for which the *generic construction* of `Check` function and the *generic hardness proof* work.

6. Decidability of Assertion Checking 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 [14] about decidability of checking validity of Herbrand equalities in presence of disequality guards. It also subsumes our earlier result [10] 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 surprising since the abstract lattice (underlying the abstractions based on convex finitary theories) often has infinite height, which implies that a standard abstract interpretation [3] based algorithm 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 [14] 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 follows.

$$\psi' = \psi \vee \texttt{Unif}(e_1 = e_2)$$

6.1.1 Correctness

The correctness of the algorithm is an easy consequence of Lemma 2 that shows that unification can be used to strengthen assertions without any loss in soundness or precision.

6.1.2 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 4. *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 4 is by establishing a well founded ordering on $\psi_i's$, and is given in Appendix E.

Lemma 4 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 4 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.)

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 $\mathbb{T}_{All} = \mathbb{T}_{LA} \cup \mathbb{T}_{UF} \cup \mathbb{T}_C \cup \mathbb{T}_{AC}$.

We use the following well-known result [2] to show that $\mathbb{T}_{All}$ is finitary.

PROPOSITION 1 ([2]). *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 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 [2].

Since equational theories are convex, the theory $\mathbb{T}_{All}$ is convex. Using these observations and Theorem 3, we can conclude that assertion checking for $\mathbb{T}_{All}$ is decidable.

7. Interprocedural Analysis

In this section, we show how to efficiently extend our assertion checking procedures to handle procedure calls. This is achieved by

computing procedure summaries that give constraints (on the input variables of the procedure) that must be satisfied for some generic assertion (involving output variables o_i of the procedure) to hold at the end of the procedure. Representation of generic assertions depends on the theory.

For example, the theory of linear arithmetic has one generic assertion that can be represented as $\alpha + \sum_i \alpha_i o_i = 0$, where α's represent unknown constants. The theory of unary uninterpreted functions has one generic assertion for each pair of variables o_1 and o_2 and can be represented as $o_1 = \alpha o_2$, where α represents an unknown sequence of unary uninterpreted functions, or equivalently, an unknown string. Procedure summaries are in the form of constraints that involve input variables and the unknown α's. For example, the summary of procedure P in Figure 6 reads as: The generic assertion $x = \alpha y$ holds at the end of procedure P iff the constraint $x = \alpha y \wedge fgx = \alpha gfy \wedge fx = \alpha fy$ holds at the beginning of procedure P. The only solution of this constraint is $\alpha \mapsto f, x \mapsto fy$. (Here x and y are both input as well as output variables of procedure P, and f and g are unary uninterpreted functions).

The key idea that we use in computing such procedure summaries is to perform a backward analysis of procedures by doing unification in presence of such unknown α's. These special variables are formally called *context variables* and the unification problem is referred to as *context unification* in the theorem proving community, where it is an active area of research [22].

In the next two sub-sections, we show how to use these ideas to build PTIME inter-procedural analyses for the unary uninterpreted abstraction, and the linear arithmetic abstraction. The former partially solves the long-standing open problem of interprocedural global value numbering, while the latter is a new proof, in our uniform setting, for a recently published result [16]. We then generalize these ideas to a class of strict unitary theories, thereby extending the results in Section 4 to inter-procedural setting. The hardness result for bitary theories (in Section 5) trivially holds in the inter-procedural setting. It remains an open challenge to see if the decidability result for finitary convex theories of Section 6 would generalize to an inter-procedural setting.

7.1 Unary Uninterpreted Functions

In this section, we describe a PTIME algorithm for inter-procedural assertion checking when the program is specified using nodes (a)-(d) and (f), and the expression language of the program involves unary uninterpreted functions. Unary uninterpreted functions can be used to model fields of structures and objects in programs. The case when the expression language of the program involves uninterpreted functions of any arity is also referred to as interprocedural global value numbering. We mention a brief history of this long-standing open problem below. The results in this section, thus, make progress towards solving this open problem.

Our results in this section use two key ideas: (a) computation of procedure summaries using unification over unknown sequences of uninterpreted functions, (b) efficient PTIME representation and manipulation of potentially exponentially large sequences using singleton context-free grammars.

7.1.1 History of Global Value Numbering

Since checking equivalence of program expressions is an undecidable problem in general, program operators are commonly abstracted as uninterpreted functions to detect expression equivalences. This form of equivalence is also called *Herbrand equivalence* [20] and the process of discovering it is often referred to as *value numbering*. Kildall [13] discovers these equivalences by performing an abstract interpretation [3] over the lattice of Herbrand equivalences in exponential time. There are several polynomial-time, but less precise, algorithms that are complete for basic blocks, but are imprecise in the presence of joins and loops in a program [1, 20, 6]. A polynomial time intraprocedural algorithm was given by Gulwani and Necula [8, 9] and Müller-Olm, Rüthing, and Seidl [14]. However, polynomial-time interprocedural global value numbering algorithm has been elusive. There are some new results, but only under severe restrictions that functions are side-effect free and one side of the assertion is a constant [18]. Neither of these assumptions is satisfied by the example in Figure 6.

7.1.2 Computing Procedure Summaries

Terms constructed using unary function symbols can be represented as strings. For example, $f(g(x))$ can be treated as the string fgx. We will denote string variables by α and concrete strings by C, D, E, F with suitable annotations.

Let $x_1, \ldots, x_n$ be all the program variables (assumed global). Consider one of the $n(n-1)$ generic assertions, say $x_1 = \alpha x_2$. We compute a summary for this generic assertion by backward propagation. This process generates a conjunction of equations of the form $Cx_i = \alpha C' x_j$ at any point in the procedure. Note that if we generate $Cx_i = \alpha C' x_j$ at the beginning of the procedure, then it simply means that there is some run (execution path) of the procedure in which $\{x_1 \mapsto Cx_i, x_2 \mapsto C'x_j, \ldots\}$. Since the number of paths is unbounded, these conjunctions can grow.

The main observation enabling summary computation is that these conjunctions can be *simplified* to contain *at most* $n(n-1)+1$ equations of the form $Cx_i = \alpha' C' x_j$—at most one equation for every pair x_i, x_j of variables except one pair, for which we can have two–and one equation of the form $\alpha = E\alpha' F$. When we perform this simplification, we replace α by $E\alpha' F$ and the summary is of the form "$x_1 = E\alpha' F x_2$ holds at the end of the procedure if some constraints over α' and the variables hold at the beginning of the procedure." See Figure 6 for an example.

The observation that we need to keep only a small number of equations $Cx_i = \alpha C' x_j$ intuitively means that we keep only a few runs. However, these runs in the *simplified* formula may not correspond to any real runs, but some equivalent hypothetical runs.

The summary computation algorithm, illustrated in Figure 6, uses the following transfer functions to compute formulas at each program point from those at successor program points in an iterative manner.

Initialization: The formula at all program points, except at procedure return, is initialized to *true*. The formula at procedure return point is initialized to be $x_1 = \alpha x_2$. (We will later repeat this computation for every pair of variables.) Thus the initial summary of the procedure is that "$x_1 = \alpha x_2$ holds at the end of the procedure if $true$ holds at the beginning."

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, that is, $\psi' = \psi[e/x]$. Note that this step preserves the *form* $Cx_i = \alpha C' x_j$ of the equations.

Non-deterministic Assignment Node: See Figure 3 (b).
If ψ is the formula after the non-deterministic assignment node $x :=?$, the formula ψ' before a non-deterministic assignment node is ψ if x does not occur in ψ, and is false otherwise.

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.

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

```
   P(){
1    while (*) { x := fgx; y := gfy; }
2    if (*) { Q(); }
3  }
   Q(){
4    while (*) { x := fx; y := fy; }
5    if (*) { P(); }
6  }
   main(){
7    y := a; x := fa; P();
8    assert(x = fy);
9  }
```

(a) Program

Ite	*Procedure P*	*Procedure Q*
0	$true$	$true$
1	$x = \alpha y, fgx = \alpha gfy$ $(\Leftrightarrow x = \alpha y, fg\alpha = \alpha gf)$	$x = \beta y, fx = \beta fy$ $(\Leftrightarrow x = \beta y, f\beta = \beta f)$
2	$x = \alpha y, fgx = \alpha gfy, fx = \alpha fy$ $(\Leftrightarrow x = \alpha y, fg\alpha = \alpha gf, f\alpha = \alpha f)$ $(\Leftrightarrow x = \alpha' fy, g\alpha' = \alpha' g, f\alpha' = \alpha' f)$ $(\Leftrightarrow x = \alpha' fy, \alpha' = \epsilon)$ $x = fy$	$x = \beta y, fx = \beta fy, fgx = \beta gfy$ $(\Leftrightarrow x = \beta y, f\beta = \beta f, fg\beta = \beta gf)$ $(\Leftrightarrow x = \beta' fy, f\beta' = \beta' f, g\beta' = \beta' g)$ $(\Leftrightarrow x = \beta' fy, \beta' = \epsilon)$ $x = fy$
3	$x = fy$	$x = fy$

(b) Summary Computation for $P()$ and $Q()$

Figure 6. This figure illustrates summary computation for interprocedural analysis over the unary abstraction. In Table (b), the summary consists of the constraints that must hold at the beginning of the procedure P (or Q) for $x = \alpha y$ (or $x = \beta y$ respectively) to be an invariant at the end of the procedure.

two branches of the conditional. However, now ψ is not of the desired form as it can have more than $n(n-1)+1$ equations of the form $Cx_i = \alpha C' x_j$. Therefore, we first call *Simplify* on it:

$$\langle \psi, \alpha = E\alpha' F \rangle = Simplify(\psi_1 \wedge \psi_2)$$

The procedure *Simplify*, described later, guarantees that (i) every solution of $\psi_1 \wedge \psi_2$ is given by $\alpha = E\alpha' F$, where α' is constrained by ψ, and (ii) ψ has at most $n(n-1)+1$ equations. Finally, we replace α by $E\alpha' F$ globally in all formulas.

Procedure Call Node: See Figure 3(f).
The formula ψ' before a procedure call node "Call $P()$" is obtained from the formula ψ after the procedure call node by using the current summary of the procedure P. If ψ is unsatisfiable, or it has a unique value for α, then computing ψ' is straight-forward. In the general case, let $Cx = \alpha C' y$ be an equation in ψ and let the summary of P, for variables x, y, be given by "$x = E\beta F y$ holds at the end of the procedure if $\bigwedge_{u,v} \bigwedge_{i=1,2} C_{uvi} u = \beta C'_{uvi} v$ holds at the beginning."

We show how to compute the weakest precondition ψ'' of $Cx = \alpha C' y$. It is clear that $Cx = \alpha C' y$ is true after the procedure call node if

$$\alpha C' = CE\beta F \ \wedge \bigwedge_{u,v} \bigwedge_{i=1,2} C_{uvi} u = \beta C'_{uvi} v$$

holds before the procedure call node. However, this is not of the required form. The equation $\alpha C' = CE\beta F$ simplifies to an equation of the form $\alpha = CE\beta F'$ or of the form $\alpha C'' = CE\beta$. In the first case, ψ'' is

$$\bigwedge_{u,v} \bigwedge_{i=1,2} C_{uvi} u = \beta C'_{uvi} v$$

with α replaced by $CE\beta F'$ globally. In the second case, ψ'' is

$$\bigwedge_{u,v} \bigwedge_{i=1,2} CEC_{uvi} u = \alpha C'' C'_{uvi} v.$$

Fixed-point Computation: In presence of loops and/or recursively defined procedures, the algorithm iterates until the formulas and/or summaries computed at each program point have the same solutions, or if any formula has no solutions. The example in Figure 6 illustrates this process.

Correctness

Lemma 1 shows that unification computation preserves invariance. Hence, for correctness, we only need to show that *Simplify* preserves all unifiers.

Let $\psi = \bigwedge_{x,y} \psi_{xy}$, where, for a fixed pair x, y of variables, ψ_{xy} contains all equations of the form $Cx = \alpha C' y$ in ψ. We show that in Lemma 5 that ψ_{xy} can be simplified to a set containing at most one equation of the form $Cx = \alpha' C' y$ and finitely many equations of the form $E\alpha' = \alpha' E'$, where $\alpha = F\alpha'$.

LEMMA 5. *The set $\psi_{xy} = \{C_i x = \alpha C'_i y : i = 1, 2, 3, \ldots, k\}$ of equations either has no solutions, or all of its solutions are given by $\alpha = F\alpha'$, where α' is constrained by a computable set of the form $\{Dx = \alpha' D' y, E_i \alpha' = \alpha' E'_i, i = 2, \ldots, k\}$.*

In the next step, the set of equations $\{E_i\alpha' = \alpha' E'_i, i = 2, \ldots, k\} \cup \{EF\alpha' = F\alpha' E'\}$ is simplified by repeated use of Lemma 6.

LEMMA 6. *The equation set $\psi = \{D_1\alpha = \alpha D'_1, D_2\alpha = \alpha D'_2\}$ is either unsatisfiable, or has a unique solution, or is equivalent to a set of the form $\psi' = \{D\alpha' = \alpha' D', \alpha = \alpha' E\}$. Moreover, there is a algorithm that computes these outcomes.*

In this way, using the above two results, any formula $\psi \equiv \bigwedge_{xy} \psi_{xy}$ can be simplified to a formula of the form $\psi' \equiv \alpha = E\alpha' F \wedge \bigwedge_{xy} \psi'_{xy}$ where ψ'_{xy} contains at most one equation of the form $Cx = \alpha' C' y$ and at most one of the form $E\alpha' = \alpha' E'$. However, $Cx = \alpha' C' y \ \wedge \ E\alpha' = \alpha' E'$ is easily seen to be equivalent to $Cx = \alpha' C' y \ \wedge \ ECx = \alpha' E' C' y$ which is of the required form. Note that the formula $E\alpha' = \alpha' E'$ does not depend on the variables x, y. Hence, ψ needs to have at most $n(n-1)+1$ equations–one for each pair of variables and one to encode $E\alpha' = \alpha' E'$ in the required form.

Termination

If we add a new equation $Cx = \alpha C' y$ to ψ while computing fixpoint across a loop, and ψ becomes unsatisfiable, then the fixpoint iterations stop immediately. If α is uniquely determined by the new equation, then the iterations stop in one more step. If the new equation is redundant, then again the iterations stop. Finally, if the new equation is not redundant and ψ is still satisfiable, then either the cardinality of ψ increases (but it is bounded by n^2+1), or, its cardinality remains the same but the size of $|E|$ (where $E\alpha = \alpha E'$ is the strongest such equation implied by ψ') is at most half of its original size. (This can be observed from the proofs of Lemma 5 and Lemma 6.) This shows that the fixpoint iterations terminate in polynomial number of steps.

Interprocedural assertion checking

Assertion checking in the interprocedural case is performed in a two phase process. In the first phase, we compute summaries

for each procedure, as described above. The final summary of a procedure P with n output variables and m input variables will consist of the following: for each ordered pair x_1, x_2 of output variables, we will have "$x_1 = E\alpha F x_2$ *if* some constraint ψ containing (at most) $n(n-1)+1$ equations holds at the beginning of the procedure." Thus, the summary for any procedure has at most n^4 equations.

In the second phase, we perform assertion checking using backward analysis starting from the *given* assertion, and using the procedure summaries computed above.

7.1.3 Efficient Representations

The *Simplify* procedure inherent in the proof of Lemma 5 and Lemma 6 is easily seen to take time polynomial in the size of the equation set ψ. However, using a naive (explicit) representation, the size of ψ can be *exponential* in the size of the program, as in the following example.

EXAMPLE 2. *Consider the n procedures $P_0, \ldots, P_{n-1}$ defined as*

$$P_i(x_i)\ \{\ t := P_{i-1}(x);\ y_i := P_{i-1}(t);\ \mathit{return}(y_i);\}$$
$$P_0(x_0)\ \{\ y_0 := f x_0;\ \mathit{return}(y_0);\}$$

The summary of procedure P_i is: $y_i = \alpha x_i$ iff $\alpha = f^{2^i}$.

Here we appeal to shared representation of strings using *singleton context-free grammars* (SCFG). Since the program itself implicitly represents the strings C_i's (in the set of equations in Lemma 5) using this shared representation, we know that such a shared representation is linear in the size of the program.

EXAMPLE 3. *Following up on Example 2, we note that the string f^{2^n} can be represented by the SCFG with start symbol A_n and productions $\{A_{i+1} \to A_i A_i : i = 1, \ldots, n\} \cup \{A_0 \to f\}$. In particular, the summaries of the procedures can be represented as: $y_i = \alpha x_i$ iff $\alpha = A_i$.*

A classic result by Plandowski [19] shows that equality of two strings represented as SCFGs can be checked in polynomial time. It is an easy exercise to see that prefix testing and largest common prefix/suffix computation can also be performed in polynomial time. Hence, the computational procedure outlined above can be implemented in polynomial time using the SCFG representation of strings. In conclusion, this shows that summaries can be computed in PTIME on the abstraction of unary symbols. We remark here that Plandowski's result has been generalized to trees [21] suggesting that it may be possible to generalize our result to the interprocedural global value numbering problem.

7.2 Linear Arithmetic

Müller-Olm and Seidl [16] showed that procedure summaries can be efficiently computed for the abstraction of linear arithmetic (with only equalities). This result has a simple proof in our framework. Since the theory of linear arithmetic is unitary, we just have to compute summaries for the generic assertion $\alpha_1 o_1 + \cdots + \alpha_n o_n + \alpha_0$, where $\alpha_0, \ldots, \alpha_n$ are regular arithmetic variables. Hence, the conjunction ψ of equations at any point in the procedure contains *linear* equations *over* the $(n+1)^2$ variables α_i and $(\alpha_i x_j)$, where $i = 0, 1, \ldots, n$ and x_j are program variables (bounded by n). We know that there can not be more than $(n+1)^2$ linearly independent (non-redundant) equations. This shows that ψ can have at most $(n+1)^2$ equations, which means that summaries are small and fixpoint iterations terminate.

As a final step, note that the coefficient can be large (since the program can encode large numbers succinctly) and hence to get a true PTIME procedure, we will have to resort to modulo arithmetic and randomization. We remark here that the proof of Müller-Olm and Seidl [16] is based on the the observation that *runs* of a procedure correspond to linear transformations and there can be only quadratic many linearly-independent transformations.

7.3 General Result for Unitary Theories

The interprocedural analyses presented in Sections 7.1 and 7.2 are instances of a more general framework for unitary theories. In a unitary theory, assertions can always be reduced to the form $\wedge_i x_i = e_i$ (Lemma 1). Hence a generic assertion can be written as $x_1 = \alpha[x_2, \ldots, x_n]$, where α is a context variable representing the unknown term structure. In the general framework, procedure summaries are computed by backward propagating these generic assertions through nodes (a)-(d) and (f) of Figure 3. This will generate conjunctions ψ of equations of the form $e_1 = \alpha[e_2, \ldots, e_n]$, where e_i's are expressions in the theory.

We can obtain a PTIME interprocedural analysis for programs using expressions from a strict unitary theory $\mathbb{T}$ if
(a) there is a PTIME simplification procedure (technically, procedure for unification in theory $\mathbb{T}$ in the presence of at most one context variable) that, given ψ, returns ψ' that has the same solutions as ψ and that has a polynomially bounded number of equations;
(b) there is a succinct representation for the expressions e_i's and the above procedure can efficiently work over this representation;
(c) any set ψ can only be strengthened a polynomially-bounded number of times.
These conditions guarantee, respectively, that summaries are small, they can be efficiently computed, and fixpoint iterations terminate quickly. These conditions are satisfied for unary symbols (Section 7.1) and linear arithmetic (Section 7.2). It is still open if they hold for arbitrary uninterpreted symbols (global value numbering).

This general framework is also applicable to two special cases considered in the literature—procedures have only one return value and no side effects [17, 18], and all assertions have one side constant [18]. In these cases, summary computation simplifies considerably since the context variable α can be effectively eliminated. This partly explains why the interprocedural extension in these cases is (almost) "free" [17].

8. Discussion

In this section, we discuss the broader significance of the results that we have presented in this paper.

8.1 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 (completeness, but not soundness) if we use unification to replace a weaker assertion by a stronger assertion. This loss in completeness is not surprising since the presence of *positive guards* can cause assertion checking to become *undecidable* for several abstractions [15, 14].

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: $\psi' \equiv \psi \vee \psi[x/y] \vee \psi[y/x]$. For rest of the program nodes, we can use the transfer functions suggested by backward propagation enhanced with unification. 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.

8.2 Backward vs. Forward Analysis

The results in this paper advance the state of the art of backward analyses by establishing new techniques for backward program analysis based on unification algorithms. 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). This assumes 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. The results in this paper show how to perform precise backward analysis over a large class of abstract domains by using unification algorithms from the corresponding logical theory.

Our algorithms for assertion checking are based on backward analysis of programs. For several of these algorithms, we can argue that they are better than forward analyses over corresponding program abstractions in terms of efficiency. This is because in order to perform precise assertion checking, a forward analysis would need 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 over which the computations need to be performed (to precisely decide the validity of the assertions) 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, for some abstractions, we can argue that forward analyses are more suitable than backward analyses. For example, abstractions that reason about pointers. Consider computing the weakest pre-condition across the assignment $*p := q$ without knowing the points-to-set for p. This will need to account for all possible aliasing scenarios, while a simple forward alias analysis might be able to establish a small points-to-set for pointer p.

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 [11]. 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 [7]).

8.3 Connections between Program Analysis and Theorem Proving

Another broader picture that emerges from this paper is the fruitful transfer of results from the theorem proving community to the world of program analysis. We have shown that forward program analysis can be made more precise and efficient by a tighter coupling of theorem proving technology [11]. 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 critical in improving backward analysis. Using *backward analysis enhanced with unification*, we showed that the unification type of a theory determines the complexity of the assertion checking problem for the corresponding abstraction.

We believe that observations such as the ones developed in this paper can lead to significant new insights into program analysis. For instance, the connection between unification and assertion checking that we established in this paper resulted in a new approach to computing precise procedure summaries for interprocedural analysis. Summary computation for procedures was easily observed to require backward propagation enhanced with context unification. This helped us partially solve the open problem of inter-procedural global value numbering (i.e., assertion checking in the presence of uninterpreted symbols). In particular, we gave a polynomial time algorithm for inter-procedural global value numbering in presence of only unary uninterpreted functions.

9. Conclusion and Future Work

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).

Extension to interprocedural assertion checking is possible by replacing regular unification by context unification to compute procedure summaries. This observation explains the interprocedural analysis algorithm for linear arithmetic and gives a new PTIME interprocedural assertion checking algorithm for the abstraction of unary uninterpreted symbols. Moreover, it also opens up new avenues for building interprocedural analysis engines using context unification procedures and shared representation for terms [21].

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] 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 W. Snyder. Unification theory. In *Handbook of Automated Reasoning*, volume I, chapter 8, pages 445–532. Elsevier Science, 2001.

[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] K. Gargi. A sparse algorithm for predicated global value numbering. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, volume 37, 5, pages 45–56. ACM Press, June 17–19 2002.

[7] S. Gulwani and N. Jojic. Program verification as inference in belief networks. Technical Report MSR-TR-2006-98, Microsoft Research, July 2006.

[8] S. Gulwani and G. C. Necula. Global value numbering using random interpretation. In *31st Annual ACM Symposium on POPL*, Jan. 2004.

[9] 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.

[10] S. Gulwani and A. Tiwari. Assertion checking over combined abstraction of linear arithmetic and uninterpreted functions. In *15th European Symposium on Programming*, volume 3924 of *LNCS*. Springer, Mar. 2006.

[11] S. Gulwani and A. Tiwari. Combining abstract interpreters. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2006.

[12] M. Karr. Affine relationships among variables of a program. In *Acta Informatica*, pages 133–151. Springer, 1976.

[13] G. A. Kildall. A unified approach to global program optimization. In *1st ACM Symposium on POPL*, pages 194–206, Oct. 1973.

[14] 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.

[15] 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.

[16] M. Müller-Olm and H. Seidl. Precise interprocedural analysis through linear algebra. In *31st ACM Symposium on POPL*, pages 330–341, Jan. 2004.

[17] M. Müller-Olm, H. Seidl, and B. Steffen. Interprocedural analysis (almost) for free. Technical Report 790, Fachbereich Informatik, Universität Dortmund, 2004.

[18] M. Müller-Olm, H. Seidl, and B. Steffen. Interprocedural herbrand equalities. In *Prog. Lang. and Syst., Proc. 14th Eur. Symp. on Programming, ESOP 2005*, volume 3444 of *LNCS*, pages 31–45. Springer, 2005.

[19] W. Plandowski. Testing equivalence of morphisms on context-free languages. In *Algorithms - ESA '94, Proc. 2nd Annual European Symp.*, volume 855 of *LNCS*, pages 460–470. Springer, 1994.

[20] O. Rüthing, J. Knoop, and B. Steffen. Detecting equalities of variables: Combining efficiency with precision. In *SAS*, volume 1694 of *LNCS*, pages 232–247, 1999.

[21] M. Schmidt-Schauß. Polynomial equality testing for terms with shared substructures. Technical Report 21, Institut für Informatik. Fachbereich Informatik und Mathematik. J. W. Goethe-Universität Frankfurt am Main, November 2005.

[22] M. Schmidt-Schauß and K. U. Schulz. Solvability of context equations with two context variables is decidable. *Journal of Symbolic Computation*, 33(1):77–122, 2002.

[23] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, pages 189–233. Prentice-Hall, 1981.

A. Proof of Lemma 1

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

LEMMA 7. *If* $\mathbb{T} \models \bigwedge_i e_i\sigma = e_i'\sigma$, *then* $\mathbb{T} \models \texttt{Unif}(\bigwedge_i e_i = e_i')\sigma$.

PROOF: Suppose $\mathbb{T} \models \bigwedge_i e_i\sigma = e_i'\sigma$. Let $\texttt{Unif}(\bigwedge_i e_i = e_i') = \sigma_1 \vee \cdots \vee \sigma_k$. 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'. For any variable x, we will show that σ makes x and $x\sigma_j$ equal, that is, we will show that $x\sigma =_{\mathbb{T}} x\sigma_j\sigma$. But in the theory $\mathbb{T}$, we have $x\sigma =_{\mathbb{T}} x\sigma_j\sigma' =_{\mathbb{T}} x\sigma_j\sigma_j\sigma' =_{\mathbb{T}} x\sigma_j\sigma$, using the facts that $\sigma =_{\mathbb{T}} \sigma_j\sigma'$ and that substitutions are idempotent. This completes the proof of the lemma. □

We now prove Lemma 1.

PROOF: We need to prove that $\texttt{Unif}(e = e')$ holds at π iff $e = e'$ holds at π. We will show that *in every run* of the program, $\texttt{Unif}(e = e')$ 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 $\mathbb{T} \models \texttt{Unif}(e_1 = e_2)\sigma$. The $\Leftarrow$ direction is trivial since $\texttt{Unif}(e_1 = e_2)$ implies $e_1 = e_2$ (in $\mathbb{T}$). The $\Rightarrow$ direction is a consequence of Lemma 7. □

B. 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 $O(n^4(T_{\texttt{Unif}}(n^2) + T_{\texttt{Valid}}(n^2)))$. □

C. Proof of Lemma 2

We first prove a useful lemma.

LEMMA 8. *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

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

The last step is a consequence of Lemma 8. 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 7 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. □

D. Proof of Lemma 3

LEMMA 9. *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 $j = 1$,

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

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

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

The other case can be obtained by replacing e by e' in the above proof. □

We use Lemma 9 to prove Lemma 3 below.

PROOF: (Lemma 3) We prove by induction on j that $\texttt{Assert}(e_j = e'_j)$ holds iff $\texttt{Assert}(\bigvee_{i=1}^{j+1} x = \alpha_i)$ 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 $\texttt{Unif}(t_j = t'_j)$ is $\bigvee_{i=1}^{j+2} x = \alpha_i$. For $j = 1$, since $\texttt{Unif}(e = e')$ is $y = z_1 \vee y = z_2$ by assumption, it follows that $\texttt{Unif}(e[x/y, \alpha_1/z_1, \alpha_2/z_2] = e'[x/y, \alpha_1/z_1, \alpha_2/z_2])$ is $x = \alpha_1 \vee x = \alpha_2$ (by variable renaming).
For the induction step, using the same argument, we observe that

$\texttt{Unif}(e[e_j/y, e'_j/z_1, e_j[\alpha_{j+2}/x]/z_2] = e'[e_j/y, e'_j/z_1, e_j[\alpha_{j+2}/x]/z_2])$
$\Leftrightarrow$ $\texttt{Unif}(e_j = e'_j) \vee \texttt{Unif}(e_j = e_j[\alpha_{j+2}/x])$ Bitary
$\Leftrightarrow$ $\texttt{Unif}(e_j = e'_j) \vee x = \alpha_{j+1}$ By Lemma 9
$\Leftrightarrow$ $(\bigvee_{i=1}^{j+2} x = \alpha_i)$ Induction hyp.

□

E. Proof of Lemma 4

PROOF: 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 $|\texttt{Unif}(\psi_i^\ell \wedge \alpha_i^j)| > |\psi_i^\ell|$. 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 $|\texttt{Unif}(\psi_i^{\ell'} \wedge \alpha_i^j)| \geq |\psi_i^{\ell'}|$ 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.

□

F. Proofs of Section 7

We prove Lemma 5 by first showing how two equations of the form $Cx = \alpha C'y$ can be simplified.

LEMMA 10. *The two equations, $C_i x = \alpha C'_i y$ for $i = 1, 2$, either have zero solutions, or exactly one solution, or all its solutions are given by $\alpha = E\alpha'$ where α' is constrained by two equations of the form $Cx = \alpha' C' y$ and $D\alpha' = \alpha' D'$.*

PROOF: Wlog assume $|C_1| \leq |C_2|$. We split the proof into cases.
(1) C_1 *is a suffix of* C_2. Let $C_2 = D_2 C_1$.

$C_1 x = \alpha C'_1 y,\ D_2 C_1 x = \alpha C'_2 y$ Initial ψ
$\Leftrightarrow$ $C_1 x = \alpha C'_1 y,\ D_2 \alpha C'_1 y = \alpha C'_2 y$ Using $C_1 x \mapsto \alpha C'_1 y$
$\Leftrightarrow$ $C_1 x = \alpha C'_1 y,\ D_2 \alpha C'_1 = \alpha C'_2$ Cancel y

If C'_1 is not a suffix of C'_2, then there are no solutions. Otherwise the second equation can be written in the form $D_2\alpha = \alpha D'$, where $C'_2 = D' C'_1$.
(2) C_1 *is not a suffix of* C_2. Let $C = lcs(C_1, C_2)$ be the *largest common suffix* of C_1 and C_2. Therefore, $C_1 = D_1 C$ and $C_2 = D_2 C$ for nonempty D_1, D_2. Now, there are two subcases.
(2a) C'_1 is not a suffix of C'_2. Let $C' = lcs(C'_1, C'_2)$. We will have $C'_1 = D'_1 C'$ and $C'_2 = D'_2 C'$ where D'_1, D'_2 are nonempty strings.

$D_1 Cx = \alpha C'_1 y,\ D_2 Cx = \alpha C'_2 y$ Initial ψ
$\Leftrightarrow$ $D_1 Cx = \alpha D'_1 C' y,\ D_2 Cx = \alpha D'_2 C' y$ $C' = lcs(C'_1, C'_2)$
$\Leftrightarrow$ $Cx = C' y,\ D_1 = \alpha D'_1,\ D_2 = \alpha D'_2$ Choice of C', D'

These equations, if satisfiable, will have at most one solution for α which can be easily computed from the above equations.
(2b) C'_1 is a suffix of C'_2. Let $C'_2 = D'_2 C'_1$.

$D_1 Cx = \alpha C'_1 y,\ D_2 Cx = \alpha C'_2 y$ Initial ψ
$\Leftrightarrow$ $D_1 Cx = \alpha C'_1 y,\ D_2 Cx = \alpha D'_2 C'_1 y$ Use $C'_2 = D'_2 C'_1$
Choose α' s.t. $\alpha' C'_2 y = Cx$ and $\alpha = D_1 \alpha'$
$\Leftrightarrow$ $Cx = \alpha' C'_1 y,\ D_2 Cx = D_1 \alpha' D'_2 C'_1 y$ Supress defn of α
$\Leftrightarrow$ $Cx = \alpha' C'_1 y,\ D_2 \alpha' = D_1 \alpha' D'_2$ Cancel $C'_1 y$

The second equation above has a solution only if D_1 is a prefix of D_2. In that case, it simplifies to $D\alpha' = \alpha' D'_2$, where $D_1 D = D_2$. This completes the proof.

□

PROOF: (Lemma 5) Sort the equation s.t. $|C_1| \leq |C_2| \leq \cdots \leq |C_k|$. We use Lemma 10 on the first two equations. If there is a unique solution, we test if it makes all other equations valid. If the first two equations have no solutions, then we are again done. If we instead get back $\{Cx = \alpha' C' y, D\alpha' = \alpha' D'\}$ and $\alpha = E\alpha'$, we replace α by $E\alpha'$ in all the other equations.

We repeat the above process now on the set $\{Cx = \alpha' C' y, C_i x = E\alpha' C_i' y, i = 3, \ldots, k\}$. Note that $|E| \leq |C_1|$ and hence this new equation set, if satisfiable, can be written in the form $\{Cx = \alpha' C' y, D_i x = \alpha' C_i' y, i = 3, \ldots, k\}$.

Since we reduce the number of equations by one in each step, after $k-1$ steps, we will be left with at most one equation of the form $Dx = \alpha' D' y$, at most $k-1$ equations of the form $E\alpha' = \alpha' E'$, and one equation $\alpha = F\alpha'$ that keeps track of the relationship of the current variable α' with the original variable α.

□

We will next prove two useful lemmas about equations over strings before proving Lemma 6.

LEMMA 11. *The equation $C\beta = \beta D$ has the same set of unifiers as the equation $C^l\beta = \beta D^l$ for all $l \geq 1$.*

PROOF: $\Rightarrow$: If $C\beta = \beta D$, then $CC\beta = C\beta D = \beta DD$ and applying this repeatedly we conclude that $C^l\beta = \beta D^l$.
$\Leftarrow$: We will show that every solution of $C^l\beta = \beta D^l$ is also a solution of $C\beta = \beta D$. Let β be a solution of $C^l\beta = \beta D^l$. First suppose that the length of β is atmost $|C|^l$. Therefore, β is a prefix of C^l that is also a suffix of D^l. Let $C^l = \beta\beta'$. Then D has to be $\beta'\beta$. Therefore, there are A, B s.t. $\beta = C^m A$, $\beta' = BC^{l-m-1}$, $C = AB$, and $D = BA$. Now $C\beta = ABC^m A = C^{m+1}A$ and $\beta D = C^m ABA = C^{m+1}A$ and hence β is a solution for $C\beta = \beta D$. Finally, if length of β is more than $|C|^l$, then necessarily $\beta = C^m\beta'$, where $m \geq l$ and length of β' is less than $|C|^l$. By previous case, we know that $C\beta' = \beta' D$. Hence, it is easy to see that $C\beta = \beta D$ as well.

□

LEMMA 12. *The equation set $\{C\beta = \beta C', CD\beta = \beta C'D'\}$ has the same unifiers as the set $\{C\beta = \beta C', D\beta = \beta D'\}$.*

PROOF: As far as the set of unifiers is concerned, note that

$$\begin{array}{lll} & C\beta = \beta C',\ CD\beta = \beta C'D' & \\ \Leftrightarrow & C\beta = \beta C',\ CD\beta = C\beta D' & \text{Replacing } \beta C' \text{ by } C\beta \\ \Leftrightarrow & C\beta = \beta C',\ D\beta = \beta D' & \text{Cancelling } C \text{ from both sides} \end{array}$$

□

PROOF: (Lemma 6) Wlog assume that $|D_1| \leq |D_2|$. Note that if $|D_1| \neq |D_1'|$ or $|D_2| \neq |D_2'|$, then ψ has no solutions. Henceforth, by convention, the length of the string represented by unprimed variable (e.g. D_2) will be always equal to the length of the string represented by the *corresponding* primed variable (D_2').

We distinguish the following cases:
(1) *D_1 is not a prefix of D_2:* In this case we should have $lcp(D_1, D_2) = \alpha lcp(D_1', D_2')$, where *lcp* returns the *largest common prefix* of its arguments. This equation can have at most one solution for α. Hence ψ has at most one solution.
(2) *D_1' is not a suffix of D_2':* Similar to the previous case, in this case we would necessarily have $lcs(D_1, D_2)\alpha = lcs(D_1', D_2')$. This equation can have at most one solution for α. Hence ψ has at most one solution.
(3) *Either D_1^l is not a prefix of D_2 or ${D_1'}^l$ is not a suffix of D_2', where $l = \lfloor |D_2|/|D_1| \rfloor$:* Using Lemma 11 $D_1\alpha = \alpha D_1'$ is equivalent to $D_1^n\alpha = \alpha D_1'$ for all $n \geq 1$. Hence this case is similar to (1) and (2) above and we will have at most one solution for α.
(4) *Either D_2 is not a prefix of D_1^{l+1} or D_2' is not a suffix of ${D_1'}^{l+1}$, where $l = \lfloor |D_2|/|D_1| \rfloor$:* Similar to case (3), there is at most one solution in this case.
(5) *All cases above do not apply.* In this case, we necessarily have $D_1 = AB$, $D_1' = B'A'$, $D_2 = (AB)^l A$ and $D_2' = A'(B'A')^l$.

In the following equivalence preserving transformations, we do not show definitions (equations of the form $\alpha = e$) to conserve space. First assume $A'B' = B'A'$.

$$\begin{array}{lll} & AB\alpha = \alpha B'A',\ (AB)^l A\alpha = \alpha A'(B'A')^l & \text{Initial } \psi \\ \Leftrightarrow & AB\alpha = \alpha B'A',\ (AB)^l A\alpha = \alpha (B'A')^l A' & \\ & \text{Using } A'B' = B'A' & \\ \Leftrightarrow & (AB)^l\alpha = \alpha (B'A')^l,\ (AB)^l A\alpha = \alpha (B'A')^l A' & \text{Lem. 11} \\ \Leftrightarrow & (AB)^l\alpha = \alpha (B'A')^l,\ A\alpha = \alpha A' & \text{Lem. 12} \\ \Leftrightarrow & AB\alpha = \alpha B'A',\ A\alpha = \alpha A' & \text{Lem. 11} \end{array}$$

If $A'B' \neq B'A'$, let $C = lcp(A'B', B'A')$ and let D be s.t. $CD = B'A'$. The equivalence preserving transformation below shows that the set of all solutions for ψ are given by $\alpha = \alpha_1 D(B'A')^{l-1}$, where α_1 is constrained by

$$\{AB\alpha_1 = \alpha_1 DC,\ A\alpha_1 = \alpha_1 A''\}$$

and A'' is a solution for $A'C = CA''$. If there is no such A'' then there is no solution.

$$\begin{array}{lll} & AB\alpha = \alpha B'A',\ (AB)^l A\alpha = \alpha A'(B'A')^l & \\ \Leftrightarrow & (AB)^l\alpha = \alpha (B'A')^l,\ (AB)^l A\alpha = \alpha A'(B'A')^l & \\ & \text{Lem. 11} & \\ \Leftrightarrow & (AB)^l\alpha_1 = \alpha C,\ \alpha = \alpha_1 D(B'A')^{l-1}, & \\ & (AB)^l A\alpha = \alpha A'(B'A')^l & \\ & \text{By choice of } \alpha_1 \text{ and } C & \\ \Leftrightarrow & (AB)^l\alpha_1 = \alpha_1 D(B'A')^{l-1}C, & \\ & (AB)^l A\alpha_1 D(B'A')^{l-1} = \alpha_1 D(B'A')^{l-1}A'(B'A')^l & \\ & \text{Supressing definition of } \alpha & \\ \Leftrightarrow & (AB)^l\alpha_1 = \alpha_1 D(B'A')^{l-1}C, & \\ & (AB)^l A\alpha_1 = \alpha_1 D(B'A')^{l-1}A'C & \\ & \text{Cancel } D(B'A')^{l-1} & \end{array}$$

Since α_1 was chosen such that $(AB)^l\alpha_1$ is a prefix of $(AB)^l\alpha_1$, it follows that the right-hand side strings should also have the same relation.

$$\begin{array}{lll} \Leftrightarrow & (AB)^l\alpha_1 = \alpha_1 D(B'A')^{l-1}C, & \\ & (AB)^l A\alpha_1 = \alpha_1 D(B'A')^{l-1}CA'' & \\ & \text{Necessarily } A'C = CA'' \text{ for some } A'' & \\ \Leftrightarrow & (AB)^l\alpha_1 = \alpha_1 D(B'A')^{l-1}C,\ A\alpha_1 = \alpha_1 A'' & \\ & \text{Using Lemma 12} & \\ \Leftrightarrow & AB\alpha = \alpha B'A',\ A\alpha_1 = \alpha_1 A'' & \\ & \text{Using definition of } \alpha \text{ and Lemma 11} & \\ \Leftrightarrow & AB\alpha_1 D(B'A')^{l-1} = \alpha_1 D(B'A')^l,\ A\alpha_1 = \alpha_1 A'' & \\ & \text{Using } \alpha = \alpha_1 D(B'A')^l & \\ \Leftrightarrow & AB\alpha_1 = \alpha_1 DC,\ A\alpha_1 = \alpha_1 A'' & \\ & \text{Cancel } D(B'A')^{l-1} & \end{array}$$

In *either* case, note that the new pair of equation derived in the last step (which is equivalent to initial equations modulo the definition α) are smaller than the original set. In fact, the sum of the lengths, $|AB| + |A|$ is at most half the sum of lengths $|AB| + |AB|^l + |A|$. Hence, the number of times this procedure can be applied repeatedly is bounded by log of the maximum length of input strings, which is bounded by 2^n. As a result, after at most n iterations, we will terminate with just one equation remaining (unless a unique solution is found or the equations are detected to be unsatisfiable at some intermediate step).

□