Logical Interpretation: Static Program Analysis Using Theorem Proving*

Ashish Tiwari[1] and Sumit Gulwani[2]

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

Abstract. This paper presents the foundations for using automated deduction technology in static program analysis. The central principle is the use of *logical lattices* – a class of lattices defined on logical formulas in a logical theory – in an abstract interpretation framework. Abstract interpretation over logical lattices, called *logical interpretation*, raises new challenges for theorem proving. We present an overview of some of the existing results in the field of logical interpretation and outline some requirements for building expressive and scalable logical interpreters.

1 Introduction

Theorem proving has been one of the core enabling technologies in the field of program verification. The traditional use of theorem proving in program analysis is based on the concept of deductive verification, where the program is *sufficiently* annotated with assertions and verification conditions are generated that, if proved, will establish that the annotated assertions are indeed invariants of the program. The generated verification conditions are discharged using a theorem prover [6]. This traditional approach of integrating theorem proving with program analysis requires theorem provers in the form of satisfiability checkers for various theories and combination of theories.

While deductive verification attempts to *verify* given annotated invariants, abstract interpretation seeks to *generate* invariants for programs. Static analysis and abstract interpretation techniques have seen recent success in the field of software verification. For example, the Slam project and the Astree tool demonstrated effectiveness of static analysis techniques in verifying large pieces of device driver and aerospace code. However, static analysis techniques can verify only simple properties. Most of the current tools based on these techniques target a specific class of programs and prove only specific kinds of properties of these programs. The working hypothesis of this paper is that theorem proving technology can push static analysis techniques to handle larger classes of properties for larger classes of systems.

* The first author was supported in part by the National Science Foundation under grant ITR-CCR-0326540.

The process of going from checking annotated invariants (deductive verification) to generating the invariants (abstract interpretation) is akin to going from type checking to type inference. While traditional theorem provers, which are essentially satisfiability solvers, can naturally help in invariant checking, how can they be extended to help in this new task of invariant generation?

Shankar [24] proposed the general paradigm of using theorem proving technology in the form of "little engines of proof". The idea was to use components of monolithic theorem provers, such as decision procedures, unification and matching algorithms, and rewriting, as embedded components in an application. This idea gained remarkable traction and lead to the development of several little engines, mostly in the form of *satisfiability modulo theory*, or SMT, solvers. Continued research and development has resulted in dramatic improvements in the performance of these little engines of proof.

Satisfiability checking procedures, while useful in the context of deductive verification, are insufficient for building abstract interpreters. This paper considers the approach of embedding theorem proving technology, in the form of little engines, as an embedded component in modern software verification tools based on abstract interpretation. This integration uses theorem proving technology in a new role and raises several interesting new questions.

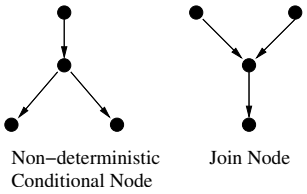
How do we formally understand this new approach of integrating theorem proving and program analysis? What little engines are required for this purpose and how can they be built? What is the interface that is required to smoothly embed a little engine inside a static analysis tool based on abstract interpretation?

This paper answers these questions by laying the foundations of *logical interpretation* – the process of performing abstract interpretation on *logical lattices*. A logical lattice is a lattice whose domain consists of formulas and whose ordering relation is (a refinement of) the logical implication relation $\Rightarrow_{\mathbb{T}}$ in some logical theory $\mathbb{T}$. We present an overview of the existing results in the area of logical interpretation and outline directions for future work.

We start by defining the assertion checking problem in Section 2. We introduce logical lattices in Section 3. In Section 4 we use logical interpretation to solve the assertion checking problem in an intraprocedural setting for various program models. Results for interprocedural analysis are presented in Section 5. In Section 6, we consider the problem of assertion generation using a (forward) logical abstract interpreter. Finally, we briefly discuss richer program models that include heaps in Section 6.2, and the interface of a logical lattice that enables modular construction of quantified logical abstract domains for reasoning about these richer program models in Section 6.3.

2 Problem Definition

We first present the program model and its semantics in Section 2.1, followed by a quick introduction to abstract interpretation in Section 2.2. Section 2.3 will formally define the assertion checking problem.



	Edge Label	Name of Edge
(a)	$x := e$	Assignment
(b)	$x :=?$	Non-det. Assgn.
(c)	`skip`	Skip
(d)	`Assume`$(e_1 \neq e_2)$	Assume
(e)	`Call`(P)	Procedure Call

Fig. 1. Conditional and join nodes in flowcharts and various types of edge labels.

2.1 Program Model

A *program* is a directed graph whose edges are labeled by one of the five labels in Figure 1. Each connected component of such a graph, called a *procedure*, is assumed to have a marked entry node with no incoming edges and a marked exit node with no outgoing edges. Each procedure is labeled with a name (of the procedure). Without loss of generality, we assume that, for each node, either its indegree or its outdegree is at most one.

The program model is parameterized by a theory $\mathbb{T}$. We assume, henceforth, that $\mathbb{T}$ is some theory over a signature Σ. Let $\mathit{Terms}(\Sigma \cup X)$ denote the set of terms constructed using signature Σ and variables X. Examples of $\mathbb{T}$ that will be used particularly in this paper are the theory of linear arithmetic, the theory of uninterpreted symbols, the combination of these theories, and the theory of commutative functions.

Let X be a finite set of program variables. The edges of a program are labeled by either (a) an assignment ($x := e$), or (b) a non-deterministic assignment ($x :=?$), or (c) a skip, or (d) an assume ($\texttt{Assume}(e_1 \neq e_2)$), or (e) a procedure call ($\texttt{Call}(P)$). Here $x \in X$, $e, e_1, e_2 \in \mathit{Terms}(\Sigma \cup X)$, and P is a name for a procedure in the program.

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 can not be precisely expressed in the simplified program model.

A *join node* is a node with two (or more) incoming edges. A *non-deterministic conditional node* is a node with two (or more) outgoing edges. This denotes that the control can flow to either branch irrespective of the program state before the conditional. Such nodes can be used as a safe abstraction of guarded conditionals that cannot be expressed precisely in the simplified program model. We assume, without loss of generality, that all (incoming or outgoing) edges incident on a join or non-deterministic conditional node are labeled with `skip`.

Assume edges, $\texttt{Assume}(e_1 \neq e_2)$, can be used to partially capture conditionals. 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. The presence of disequality assume edges allows us to capture the false branch precisely in this case. Assume labels of the form $\texttt{Assume}(e_1 = e_2)$ are disallowed in the simplified model.

Our definition of a program is quite restrictive. The extension to richer program models is briefly discussed in Section 6.2. Nevertheless, this simplied program model is also useful for program analysis. It is used by abstracting a given original program using only the edge labels shown in Figure 1 and then using the results described in this paper on the abstracted program. A second motivation for studying the simplified program model is to study the theoretical aspects of static program analysis and characterize the "precision" of static program analysis that is achievable using abstract interpretation techniques.

Semantics of Programs The semantics of programs in the above program model is given in the standard way. Let *States* denote the set of all possible mappings from X to $Terms(\Sigma)$. In other words, *States* is the set of ground substitutions. Let $(Pow(States), \subseteq)$ be the complete lattice[1] defined over the domain $Pow(States)$, which is the collection of all subsets of *States*, by the usual set operators. Let $Pow(States) \mapsto Pow(States)$ denote the collection of all functions from $Pow(States)$ to $Pow(States)$. The preorder in the lattice $(Pow(States), \subseteq)$ induces a (lattice) preorder on the domain $Pow(States) \mapsto Pow(States)$.

The semantics of each node in the program, say π, is given as a function $[\![\pi]\!]$ in $Pow(States) \mapsto Pow(States)$. Intuitively, $[\![\pi]\!](\psi)$ is the set of program states reached at program point π starting from some program state in ψ at the entry point of the procedure containing π. Formally, if $\boldsymbol{\pi}$ is the set of all program points in a program, then the vector of functions $([\![\pi]\!])_{\pi \in \boldsymbol{\pi}}$ is obtained as the least fixpoint of a set of fixpoint equations obtained from the program (using the semantics of the edge labels); that is, $([\![\pi]\!])_{\pi \in \boldsymbol{\pi}} = F(([\![\pi]\!])_{\pi \in \boldsymbol{\pi}})$, where F is the monotone function representing the strongest post-condition transformer. Note that for a join node π with parents π_1 and π_2, $[\![\pi]\!]$ is the join (union) of $[\![\pi_1]\!]$ and $[\![\pi_2]\!]$. See [4] for further details.

2.2 Abstract Interpretation

Abstract interpretation [4] is a generic framework for describing several static program analyses. The idea is to evaluate the program semantics over more *abstract* or *simpler* lattices, rather than the lattices over which the semantics is defined. Thus, abstract interpretation is parameterized by the choice of the abstract lattice. Its effectiveness is dependent on the expressiveness of the abstract lattice and the computational complexity of computing fixpoints in that lattice.

Let $(A, \sqsubseteq)$ be a complete lattice. Such a lattice will be called an *abstract* lattice if there is a Galois connection between $(A, \sqsubseteq)$ and $(Pow(States), \subseteq)$ defined by the monotone abstraction function $\alpha : Pow(States) \mapsto A$ and the monotone concretization function $\gamma : A \mapsto Pow(States)$. Abstract interpretation involves solving the fixpoint equations defining the semantics of the program, but over such an *abstract lattice* $(A, \sqsubseteq)$ (and the induced extension on

[1] A lattice $(A, \sqsubseteq)$ is identified by its domain A and its preorder $\sqsubseteq$. The meet ($\sqcap_A$) and join ($\sqcup_A$) are defined as the greatest lower bound and least upper bound respectively. The top and bottom elements are denoted by $\top_A$ and $\bot_A$.

$A \mapsto A$), rather than on the concrete lattice $Pow(States)$. Formally, $(\llbracket \pi \rrbracket_A)_{\pi \in \boldsymbol{\pi}} = F_A((\llbracket \pi \rrbracket_A)_{\pi \in \boldsymbol{\pi}})$, where F_A is the strongest (post-condition) transformer on the abstract domain such that $F \subseteq \gamma \circ F_A \circ \alpha$. Note that, since the lattice $(A, \sqsubseteq)$ is assumed to be complete, fixpoints exist (without requiring any widening to achieve convergence), and $\llbracket \pi \rrbracket_A$ are well-defined.

Given an abstract domain $(A, \sqsubseteq)$, the *interprocedural* abstract interpretation problem seeks to compute $\llbracket \pi \rrbracket_A$ for each program point π. If we consider a program model where there are no edges labeled by $\texttt{Call}(P)$, then we obtain the simpler problem of *intraprocedural* analysis. In the case of intraprocedural analysis, we can work on the lattice $(A, \sqsubseteq)$ directly by just focusing on computing $\llbracket \pi \rrbracket_A(\top_A)$.

We remark here that the requirement that $(A, \sqsubseteq)$ be a complete lattice can be relaxed. Abstract interpretation based analysis can be performed on semi-lattices A that are not closed under (arbitrary) join. In such a case, the join is over-approximated by some element of the lattice A. However, if we insist that there is a unique least fixpoint solution of the semantic equations in A, then we need to assume that A is a complete lattice.

2.3 The Assertion Checking Problem

We now define the assertion checking problem for programs.

Definition 1 (Assertion Checking Problem). *Let $\mathbb{T}$ be a theory and P be a program using the expression language of $\mathbb{T}$. Given an assertion $e_1 = e_2$ at a program point π in P, the assertion checking problem seeks to determine if $e_1\sigma =_{\mathbb{T}} e_2\sigma$ for every substitution σ in $\llbracket \pi \rrbracket(\top)$.*

In general, an assertion ϕ can be any set of states, $\phi \in Pow(States)$, and then the assertion checking problem seeks to determine if $\llbracket \pi \rrbracket(\top) \subseteq \phi$. If the assertion checking problem has a positive solution, we say the assertion *holds* at the given program point.

The term $e_1\sigma$ is just the evaluation of e_1 along some path in the program. Thus, assertion checking problem essentially seeks to determine if e_1 and e_2 are equal at the end of every path to program point π.

Example 1. Consider the program P containing two sequential assignments: $\{\pi_0; x := f(z); \pi_1; y := f(z); \pi_2\}$, where $X = \{x, y, z\}$ are program variables and the signature $\Sigma = \{f, a\}$ contains an uninterpreted function symbol f and constant a. We are interested in checking the assertion $x = y$ at program point π_2. Now, $\llbracket \pi_2 \rrbracket(\top)$ is the set containing all ground instances of the substitution $\sigma := \{x \mapsto fz, y \mapsto fz\}$. Clearly, $x\sigma = y\sigma$ in the theory of uninterpreted symbols, and hence, $x = y$ holds at π_2.

Since computing $\llbracket \_ \rrbracket$ is often intractable, the assertion checking problem is often solved using abstract interpretation over some abstract domain. However, this can lead to incompleteness.

Example 2. Following up on Example 1, let Φ denote the set of all finite conjunctions of all variable equalities. The relation $\Rightarrow_{\texttt{EQ}}$, where EQ is the pure theory of equality, induces a natural ordering on Φ. This ordering defines a lattice structure $(\Phi, \Rightarrow_{\texttt{EQ}})$. Note that $[\![\pi_2]\!]_\Phi = [\![\pi_1]\!]_\Phi = \mathit{true}$. This is because no variable equality holds at point π_1. Even though $x = y$ holds at π_2, the lattice Φ is not expressive enough to prove it. However, if Φ contains all (conjunctions of) equalities on $\Sigma \cup X$, then $[\![\pi_2]\!]_\Phi$ will be $x = y \wedge y = f(z)$.

An abstract lattice $(A, \sqsubseteq)$, with corresponding concretization function γ, is *sufficiently expressive* for assertion checking if for every element $\phi \in A$ such that the assertion $\gamma(\phi)$ holds at point π in a program P, it is the case that $[\![\pi]\!]_A \sqsubseteq \phi$.

The following result, which is central to many results in this paper, relates unification [2] and program analysis for the simplified program model. The notation $\texttt{Unif}(E)$, where E is some conjunction of equalities, denotes the formula that is a disjunction of all unifiers in some complete set of unifiers for E.

Lemma 1 ([10]). *Let $\mathbb{T}$ be a convex finitary theory and P be a program built using edge labels (a)–(e) from Figure 1 and using the expression language of $\mathbb{T}$. Let π be a program point in P and let ϕ_i be some conjunction of equalities. Then, $\bigvee_i \phi_i$ holds at π iff $\bigvee_i \texttt{Unif}_{\mathbb{T}}(\phi_i)$ holds at π.*

We will study the assertion checking problem for different choices of the program model. Our approach will be based on using an appropriate choice of the abstract lattice A and using abstract interpretation over A.

3 Logical Interpretation

We now introduce a class of abstract lattices called logical lattices. Abstract interpretation over logical lattices will be used as an approach to solve the assertion checking problem (Sections 4 and 5) and the invariant generation problem (Section 6). The definition below is a generalization of the definition in [9].

Let $\mathbb{T}$ be a theory over signature Σ and Φ be a class of formulas over $\Sigma \cup X$. We shall assume that $\bot$ and $\top$, representing *false* and *true* respectively, are present in Φ. For example, Φ could be the set of conjunctions of atomic formulas over $\Sigma \cup X$.

Definition 2 (Logical Lattice). *A (semi-, complete) lattice $(A, \sqsubseteq)$ is a logical (semi-, complete) lattice over some theory $\mathbb{T}$ if the domain A is Φ and the partial order $\sqsubseteq$ is contained in the implication relationship $\Rightarrow_{\mathbb{T}}$ in theory $\mathbb{T}$, i.e., if $E \sqsubseteq_A E'$ then $E \Rightarrow_{\mathbb{T}} E'$.*

We first note that a complete logical lattice is an *abstract* lattice. This fact is demonstrated by the concretization function, $\gamma(E) := \{s \in \mathit{States} \mid s \models E\}$, and the abstraction mapping, $\alpha(\boldsymbol{\sigma}) := \sqcap\{E \mid \forall s \in \boldsymbol{\sigma}.s \models E\}$ (since A is assumed to be a complete lattice, this is well-defined). Note that the concretization function, γ, is monotone, since $E \sqsubseteq E'$ implies $E \Rightarrow_{\mathbb{T}} E'$ (by definition), which in turn implies $\gamma(E) \subseteq \gamma(E')$.

In this paper, we will mainly use logical lattices in which $\sqsubseteq$ is equal to $\Rightarrow_{\mathbb{T}}$. The first part of this paper will focus on intraprocedural analysis, whereas the second part will consider interprocedural analysis.

4 Intraprocedural Logical Interpretation

In this section we use logical interpretation to solve intraprocedural assertion checking problem. We shall consider different assertion checking problems parameterized by (a) the theory $\mathbb{T}$ which interprets the expressions e that occur in the assignment and assume statements in the program model and (b) the language of the assertions.

4.1 The Theory of Uninterpreted Symbols

Let Σ be a finite set of uninterpreted function symbols and constants. Let $\mathbb{T}_{\mathtt{UFS}}$ be the theory of uninterpreted function symbols. Let X be a set of program variables. We are given a program with edge labels (a)–(c) from Figure 1, where the expression e is any term in $\mathit{Terms}(\Sigma \cup X)$. We are interested in checking assertions of the form $x = e$ at some program point π. Informally, the problem is to determine if the assertion $x = e$ evaluates to true on each program path.

We use a logical lattice defined by the theory $\mathbb{T}_{\mathtt{UFS}}$ to solve this assertion checking problem. The class Φ of formulas we consider is the set of finite *substitutions*, that is, Φ contains formulas of the form, $\bigwedge_i x_i = t_i$, such that $t_i \in \mathit{Terms}(\Sigma \cup X \mid_{\prec x_i})$ where $\prec$ is some total order on the program variables x_i's. We first observe that we can define a lattice over this choice of Φ.

Proposition 1 ([13]). *The set Φ under the $\Rightarrow_{\mathtt{UFS}}$ forms a lattice. If X is finite, then this is a complete lattice.*

The following theorem states that the assertion checking problem is decidable in polynomial time.

Theorem 1 ([7, 10]). *Let $\mathcal{P}$ denote the class of programs built using edge labels (a)–(c) and using the expression language $\mathit{Terms}(\Sigma \cup X)$, where X is a finite set of program variables. Let Φ be as defined above. The assertion checking problem for programs in class $\mathcal{P}$ and assertions in Φ is solvable in PTIME .*

Gulwani and Necula [7] presented a polynomial-time *forward* abstract interpreter over the lattice $(\Phi, \Rightarrow_{\mathtt{UFS}})$ to prove Theorem 1. A naive forward interpreter was shown to blow-up the size of generated facts, and hence, Gulwani and Necula had to prune large facts that were not *relevant* to the assertion. This process of making the forward interpreter *goal-directed* was crucial in proving the above PTIME result. Prior to that, there were other either incomplete, or exponential, procedures to solve the above assertion checking problem [1, 23]. All these procedures were based on forward abstract interpretation. It should be noted that to prove completeness of forward interpreter for the assertion checking problem, one needs to show that the above abstract domain is *sufficiently expressive*.

Gulwani and Tiwari [10] provided a new proof of a result that is more general than Theorem 1. In contrast to earlier procedures, the result in [10] is based on *backward* propagation. Backward propagation is naturally goal-directed. Unfortunately, backward propagation through assignment edges may create equations that are *not* of the form $x = e$ anymore. This is illustrated below.

Example 3. Let P be $\{\pi_0; x := a; y := f(a); \pi_1; \mathit{while}(*)\{\pi_2; x := f(x); y := f(y); \}; \pi\}$. Suppose we wish to check the assertion $y = f(x)$ at point π. We perform a backward propagation in the logical lattice $(\Phi, \Rightarrow_{\texttt{UFS}})$ defined above. To prove $y = f(x)$ at π, we need to prove $fy = ffx \wedge ffy = fffx \wedge \cdots$ at π_1. This conjunction is not finite and is not in Φ.

Both these problems are solved by Lemma 1, which notes that $e_1 = e_2$ can be replaced by $\texttt{Unif}(e_1 = e_2)$ during the backward propagation without any loss of soundness (or completeness) [8, 10]. This converts arbitrary $e_1 = e_2$ into the form of equalities in Φ. Moreover, it also helps to show that the conjunction will be finite and fixpoint is reached in n steps. The soundness of Lemma 1 provides an alternate proof of the fact that the lattice $(\Phi, \texttt{UFS})$ is sufficiently expressive for the assertion checking problem described above.

Example 4. By applying `Unif` to the assertion $fy = ffx$, we get $y = fx$ as the assertion at π_1. Thus, in Example 3, backward propagation enhanced with unification gives the assertion $y = fx$ at π_1 and π_2, and *true* at π_0 - thus proving that $y = fx$ holds at π.

The backward procedure is now seen to terminate in PTIME since the most-general unifier of a set of equations contains atmost n equations and a substitution can be strengthened at most n times.

If we enrich the programming model to include assume edges, $\texttt{Assume}(e_1 = e_2)$, then the problem of assertion checking can be easily shown to be undecidable [16]. This is partly the reason why we do not consider guarded conditionals in our program model. Coincidentally, Lemma 1 fails to hold in the presence of such assume edges.

Example 5. Consider the program $\{\texttt{Assume}(fx = fy); \pi\}$ with just one assume edge. It is evident that $fx = fy$ holds at π, but it is clear that $x = y$, which is $\texttt{Unif}(fx = fy)$, does not hold at π.

4.2 The Theory of Linear Arithmetic

Let Σ be the signature of linear arithmetic (without inequality predicates). Let X be the program variables. Let $\mathbb{T}_{\texttt{LAE}}$ denote the theory of linear arithmetic equalities. Let Φ be the class of formulas containing all finite conjunction of linear equations of the form, $\sum_{x \in X} c_x x = c$, where c_x, c are integer constants.

We are given a program with edge labels (a)–(c) from Figure 1, where the expression e is any term in $\mathit{Terms}(\Sigma \cup X)$. We are also given an assertion from the set Φ at some program point π. The following theorem states that this assertion checking problem is decidable in polynomial time.

Theorem 2 ([10]). *Let $\mathcal{P}$ denote the class of programs built using edge labels (a)–(c) and using the expression language Terms($\Sigma \cup X$), where X is a finite set of program variables. Let Φ be as defined above. The assertion checking problem for programs in class $\mathcal{P}$ and assertions in Φ is solvable in PTIME .*

The approach to solving this problem is based on using logical interpretation on the lattice $(\Phi, \Rightarrow_{\texttt{LAE}})$. Karr [14] presented a forward abstract interpreter over this logical lattice. To prove Theorem 2 using a forward interpreter requires that we also show that this abstract domain is sufficiently expressive.

A simple backward propagation algorithm gives a simple proof of Theorem 2. It is easy to see that Φ is closed under backward propagation through assignment edges. This shows that the above logical lattice is sufficiently expressive. Termination of backward propagation follows from the observation that there can be at most n linearly independent (linear) equations. We note here that we do not need to explicitly use unification since it is inbuilt in the theory `LAE`.

We remark here that the assertion checking problem becomes undecidable if we include assume edges, $\texttt{Assume}(e_1 = e_2)$, in the program model [17].

Theorem 1 and Theorem 2 give the complexity of the decision version of the problem of abstract interpretation over logical lattices defined by the theories `UFS` and `LAE`.

4.3 Unitary Theory

Theorem 1 and Theorem 2 are special cases of a more-general theorem for unitary theories. Let $\mathbb{T}$ be a unitary theory defined over a signature Σ. Let X be the program variables. The class Φ of formulas consists of conjunctions representing substitutions.

Theorem 3 ([10]). *Let $\mathbb{T}$ be a unitary theory. Assume that for any sequence of equations $e_1 = e'_1, e_2 = e'_2, \ldots$, the sequence of most-general unifiers $\texttt{Unif}(e_1 = e'_1), \texttt{Unif}(e_1 = e'_1 \wedge e_2 = e'_2), \ldots$ contains at most n distinct unifiers where n is the number of variables in the given equations. 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.* [2] *Then the assertion checking problem for programs of size n that are specified using edge labels (a)-(c) and whose expressions are from theory $\mathbb{T}$, can be solved in time $O(n^4 T_{\texttt{Unif}}(n^2))$.*

The proof of this theorem can be obtained by generalizing the proofs of Theorem 1 and Theorem 2. Specifically, we perform backward propagation on the logical lattice $(\Phi, \Rightarrow_{\mathbb{T}})$. Using unification (Lemma 1), the generated intermediate assertions are always strengthened to elements of Φ. This shows that we are essentially doing an (backward) abstract interpretation over the logical lattice $(\Phi, \Rightarrow_{\mathbb{T}})$. The assumption in the statement of Theorem 3 guarantees that fixpoint is reached in n iterations. Thus, Theorem 3 characterizes the complexity of (the decision version of) the abstract interpretation problem over certain logical lattices induced by unitary theories.

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

4.4 Checking disequality assertions is intractable

It is natural to wonder about the complexity of checking a disequality assertion in a program containing restricted kinds of edges. Unfortunately, it is easy to see that this problem is undecidable even for the simple case of programs using only uninterpreted symbols.

Theorem 4. *Checking disequality assertions is undecidable for programs with edge labels (a)-(c) and whose expression language is restricted to unary uninterpreted function symbols and one constant.*

Proof. Given an instance $\{(u_i, v_i) : i = 1, \ldots, n\}$ of PCP, consider the program:

```
1 u := ε; v := ε;
2 switch (*)
3         case 1: u := u1(u); v := v1(v);
4         case 2: u := u2(u); v := v2(v);
5         ⋮
6         case n: u := un(u); v := vn(v);
7 if (*) then { goto line number 2 } else { }
```

This program can be implemented using only edge labels (a)-(c). It nondeterministically generates (encodings of) all possible pairs (u, v) of strings such that $u = u_{i_1} \ldots u_{i_k}$ and $v = v_{i_1} \ldots v_{i_k}$. The disequality $u \neq v$ holds at the end of the program iff the original PCP does not have a solution.

4.5 Checking disjunctive assertions

We briefly consider the problem of checking if disjunctive assertions of the form $e_1 = e_2 \vee e_3 = e_4 \vee \cdots$ hold at program points. The following simple program shows that this problem is coNP-hard even for programs over a very simple expression language containing just two distinct (uninterpreted) constants.

```
IsUnSatisfiable(ψ)
% Suppose ψ has n variables x1,...,xn and m clauses # 1,...,m
% Suppose xi occurs in positive form in clauses # Ai[0],...,Ai[ci]
%              and in negative form in clauses # Bi[0],...,Bi[di].
for i = 1 to m do
    ei := 0; % ei represents whether clause i has been satisfied.
for i = 1 to n do
    if (*) then % set xi to true
        for j = 0 to ci do e_{Ai[j]} := 1;
    else % set xi to false
       for j = 0 to di do e_{Bi[j]} := 1;
Assertion(e1 = 0 ∨ e2 = 0 ∨ ··· ∨ em = 0);
```

We note that the above program can be easily written as a program using edge labels (a)-(c) by simply unrolling the loop and converting it into a loop-free program. It is easy to see that in the above program, the disjunctive assertion at the end of the program holds iff the given 3-CNF formula is unsatisfiable.

Theorem 5. *The problem of checking disjunctive equality assertions in programs with edge labels (a)-(c) and over an expression language containing (at least) two constants is coNP-hard.*

4.6 The combined theory of UFS+LAE

We now consider the problem of assertion checking (equality assertions) in programs whose expression language comes from the union of UFS and LAE theories. This problem can be shown to be coNP-hard [8] by creating a program – very similar to the program in Section 4.5 – whose expression language is that of UFS+LAE and in which an equality assertion is valid iff a given 3-CNF formulas is unsatisfiable. A crucial component of the coNP-hardness proof is the fact that an *equality* assertion in UFS+LAE can encode a *disjunctive* assertion. Specifically, note that $x = a \vee x = b$ can be encoded as the (non-disjunctive) assertion $F(a) + F(b) = F(x) + F(a + b - x)$. This idea can be generalized to encode $x = a_1 \vee x = a_2 \vee x = a_3 \vee \cdots$. Note that, by Lemma 1, $Fa+Fb = Fx+F(a+b-x)$ holds at a program point π iff $\texttt{Unif}(Fa+Fb = Fx+F(a+b-x))$ holds at π. In the theory $\texttt{UFS}+\texttt{LAE}$, $\texttt{Unif}(Fa+Fb = Fx+F(a+b-x))$ is just $x = a \vee x = b$. By recursively using this same idea, we can find an equation whose complete set of unifiers is a disjunction of the form $x = a_1 \vee x = a_2 \vee \cdots \vee x = a_n$. This observation, combined with Theorem 5, proves the following result.

Theorem 6 ([8]). *Let $\mathcal{P}$ denote the class of programs built using edge labels (a)–(c) and using the expression language $Terms(\Sigma \cup X)$, where Σ is the signature of $\texttt{UFS}+\texttt{LAE}$ and X is a finite set of program variables. The problem of checking equality assertions for programs in class $\mathcal{P}$ is coNP-hard.*

4.7 Bitary Theories

The proof of coNP-hardness of assertion checking on programs whose expression language comes from UFS+LAE can be generalized to a class of non-unitary theories that can encode the disjunction $x = a \vee x = b$ as (the complete set of) unifiers of some equality.

Specifically, 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 also require 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 $y = y' \wedge z_1 = z_1'$. It is easy to see that UFS+LAE is a bitary theory. The proof of Theorem 6 can be generalized for any bitary theory.

Theorem 7 ([10]). *Let $\mathbb{T}$ be a bitary theory over signature Σ. Let $\mathcal{P}$ denote the class of programs built using edge labels (a)–(c) and using the expression language $Terms(\Sigma \cup X)$, where X is a finite set of program variables. The problem of checking an equality assertion for programs in class $\mathcal{P}$ is coNP-hard.*

Some examples of bitary theories are the theories of a commutative function, combination of linear arithmetic and a unary uninterpreted function, and combination of two associative-commutative functions [10].

4.8 Revisiting the combined theory of UFS+LAE

We revisit the problem of checking equality assertions for programs built using expressions from UFS + LAE. The fact that there is no single most-general unifier of an equation in UFS+LAE suggests that the logical lattice defined over the domain of substitutions by $\Rightarrow_{\texttt{UFS+LAE}}$ is not sufficiently expressible for assertion checking (of $x = e$ assertions) in programs over UFS + LAE. One option would be to consider a more general lattice whose domain elements are conjunctions of arbitrary equations $(e_1 = e_2)$. Unfortunately, the ordering relation $\Rightarrow_{\texttt{UFS+LAE}}$ does not induce a lattice structure over this domain. This fact is already true for UFS [13] and is illustrated in two examples below.

Example 6. If Φ is the set of finite conjunctions of ground equations, then there is no *least upper bound* element $\phi \in \Phi$ such that

$$x = y \Rightarrow_{\texttt{UFS}} \phi$$
$$fx = x \wedge fy = y \wedge gx = gy \Rightarrow_{\texttt{UFS}} \phi$$

The proof of this claim can be found in [13].

Example 7. If Φ is the set of finite conjunctions of ground equations on the signature of UFS + LAE, then there is no *least upper bound* element $\phi \in \Phi$ such that

$$x = a \wedge y = b \Rightarrow_{\texttt{UFS+LAE}} \phi$$
$$x = b \wedge y = a \Rightarrow_{\texttt{UFS+LAE}} \phi$$

The reason for the nonexistence of a least upper bound $\phi \in \Phi$ is that any such ϕ would have to imply the infinite set of facts $C[x] + C[y] = C[a] + C[b]$, where $C[\_]$ is an *arbitrary* context.

As observed before, this is not a serious obstacle for assertion checking in programs defined using only uninterpreted symbols – as the logical lattice defined over the domain of substitutions, rather than conjunctions of arbitrary equations, is still sufficiently expressive with respect to assertion checking. However, in the case of programs containing symbols from UFS and LAE, this forces us to search for a logical lattice defined over a domain that is more general than just substitutions, and less general than conjunctions of arbitrary equations. A natural choice is the disjunction of substitutions.

Let Φ be the set of formulas that are disjunctions of substitutions. Let $\Rightarrow_{\texttt{UFS+LAE}}$ be an ordering relation on Φ. It is easy to see that $\Rightarrow_{\texttt{UFS+LAE}}$ induces a lattice structure on Φ. We can show that the assertion checking problem is decidable on this logical lattice, even for program models that include edge label (d) from Figure 1.

Theorem 8 ([8]). *Let $\mathcal{P}$ denote the class of programs built using edge labels (a)–(d) and using the expression language Terms($\Sigma \cup X$), where Σ is a signature*

containing uninterpreted symbols and linear arithmetic symbols, and X is a finite set of program variables. Let Φ be the set of disjunctions of substitutions. The assertion checking problem for programs in class $\mathcal{P}$ and assertions in Φ is decidable.

Proof. (Sketch) The procedure works by backward propagating formulas in Φ through the flowchart nodes. Due to Lemma 1, we can use unification to strengthen the formulas at each point. Since the UFS+LAE theory is finitary (every equation has a finite complete set of unifiers), it follows that each intermediate assertion obtained in the backward propagation can be converted to a formula in Φ. The remaining part is showing termination (of fixpoint across loops). Note that when a formula $\phi_1 \vee \cdots \phi_k$ in Φ is strengthened by conjuncting with another formula in Φ, then, in the result ϕ', for each $i \in \{1, \ldots, k\}$, it is either the case that (1) ϕ_i appears as a disjunct in ϕ', or (2) ϕ_i gets strengthened in multiple different ways and these strictly stronger forms appear as disjunct in ϕ'. Note that the strictly stronger forms necessarily instantiate some more of the *finitely many* program variables. If it is case (1) for all i, then this indicates that we have reached a fixpoint. If not, then we can see that we are smaller in some appropriately defined multiset extension of the well-founded ordering $>$ on natural numbers.

4.9 Convex and Finitary Theories

The proof of Theorem 8 depends on two critical ingredients: (1) Unification can be used to replace an arbitrary equation by a formula in Φ without compromising soundness; and (2) Unification always returns a finite complete set of unifiers. Property (1) and (2) can be shown to hold for any convex finitary theory. The combined theory of UFS and LAE is just one example of a convex and finitary theory.

Theorem 9 ([10]). *Let $\mathbb{T}$ be a convex finitary theory over signature Σ. Let $\mathcal{P}$ denote the class of programs built using edge labels (a)–(d) and using the expression language Terms$(\Sigma \cup X)$, where X is a finite set of program variables. Let Φ be the set of disjunctions of substitutions. The assertion checking problem for programs in class $\mathcal{P}$ and assertions in Φ is decidable.*

The (rich) theory obtained by combining (some or all) of the theories of linear arithmetic, uninterpreted functions, commutative functions, associative-commutative functions is finitary and convex. Hence, Theorem 9 shows that the assertion checking problem is decidable for programs that contain symbols from this large class of theories.

5 Interprocedural Logical Interpretation

In this section, we study the assertion checking problem for program models that additionally contain procedure call edges. Interprocedural analysis is considerably more difficult than intraprocedural analysis [21]. A modular way to do

interprocedural analysis is by means of computing *procedure summaries* [25]. A summary of procedure P is an abstraction $[\![P]\!]_A$ of its meaning $[\![P]\!]$ (which is a mapping from the subsets of input states to subsets of output subsets) in some abstract domain A.

5.1 The Theory of Linear Arithmetic

We first consider the logical lattice defined by the theory of linear arithmetic (LAE). The abstract domain Φ is the set of conjunctions of linear arithmetic equalities of the form $\sum_{i=1}^{n} c_i x_i = c$, where c_i, c are integer constants and x_i's are program variables. The ordering relation is $\Rightarrow_{\texttt{LAE}}$. A summary of a procedure in this abstract lattice would be a mapping from Φ to Φ.

Since the set Φ has an infinite number of elements, we can not hope to enumerate Φ and compute this abstract mapping. Consider the *generic* equation $\sum_{i=1}^{n} \alpha_i x_i = \alpha_0$, where α_i's are *variables*. The important property of the equation $\sum_{i=1}^{n} \alpha_i x_i = \alpha_0$ is that every element in Φ can be obtained by appropriately instantiating the α_i's by integer constants. Thus, a symbolic representation of the abstract summary can be obtained by backward propagating such a generic equation.

Backward propagation of a generic assertion, $\sum_{i=1}^{n} \alpha_i x_i = \alpha_0$, will result in (a conjunction of) equation of the general form,

$$\sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} \alpha_i x_j + \sum_{i=1}^{n} d_i \alpha_i = \alpha_0.$$

This can be seen as a *linear* equation *over* $n^2 + n + 1$ variables: n^2 variables representing the unknown product terms $\alpha_i x_j$ and $n + 1$ variables representing the unknown terms α_i. Since there can be at most n^2+n+1 linearly independent equations of the above form, the backward propagation computation would reach fixpoints in polynomial number of steps. This observation is the main ingredient in the proof of the following result.

Theorem 10 ([18, 11]). *Let* LAE *be the theory of linear arithmetic and Σ be its signature. Let $\mathcal{P}$ denote the class of programs built using edge labels (a)–(c),(e) and using the expression language Terms$(\Sigma \cup X)$, where X is a finite set of program variables. Let Φ be the set of conjunctions of linear equations over X. The assertion checking problem for programs in class $\mathcal{P}$ and assertions in Φ is solvable in polynomial time, assuming that the arithmetic operations can be done in $O(1)$ time.*

5.2 The Theory of Unary Uninterpreted Symbols

Consider the logical lattice defined by the theory of *unary* uninterpreted symbols (UUFS). The abstract domain Φ we used before is the set of substitutions, that is, conjunctions of equations of the form $x_i = t_i$, where t_i is a term (not containing

x_i). The ordering relation is $\Rightarrow_{\texttt{UUFS}}$. A summary of a procedure in this abstract lattice would be a mapping from Φ to Φ.

Again, the set Φ has an infinite number of elements, and hence we can not hope to enumerate Φ and compute this abstract mapping. Since we assume only unary symbols in the signature, the term t_i above can be seen as a string α (of unary symbols) applied to a program variable x_j or a designated constant ϵ. In other words, t_i can be seen as αx_j or $\alpha\epsilon$.

Consider the $n(n-1)$ *generic* equations $x_i = \alpha_{ij} x_j$, for $i \neq j \in \{1, \ldots, n\}$, along with the n *generic* equations $x_i = \beta_i \epsilon$, for $i \in \{1, \ldots, n\}$, where α_{ij}'s and β_i's are *string variables*. The important property of these total n^2 equations is that every equation in Φ can be obtained by appropriately instantiating one of the α_{ij}'s or β_i's by an appropriate string. Thus, a symbolic representation of the abstract summary can be obtained by backward propagating each of these n^2 generic equations.

Backward propagation of a single generic assertion, $x_i = \alpha_{ij} x_j$, will result in (a conjunction of) equations of the general form,

$$C x_k = \alpha_{ij} D x_l,$$

where C, D are concrete strings over Σ and x_k, x_l are either some program variables or ϵ. A technical lemma is now required to show that any conjunction of such equations can be simplified to contain at most a quadratic (in n) number of equations. The simplification procedure is only required to preserve the unifiers (and not preserve logical equivalence). A side-effect of the proof of the technical lemma shows that fixpoints are reached in quadratic number of iterations. Putting all these observations together, we get a proof of the following result.

Theorem 11 ([11]). *Let* UUFS *be the theory of unary uninterpreted symbols and Σ be its signature. Let $\mathcal{P}$ denote the class of programs built using edge labels (a)–(c),(e) and using the expression language Terms$(\Sigma \cup X)$, where X is a finite set of program variables. Let Φ be the set of substitutions over X. The assertion checking problem for programs in class $\mathcal{P}$ and assertions in Φ is solvable in polynomial time, assuming that string operations can be done in $O(1)$ time.*

All string operations that arise in the proof of Theorem 11 can indeed be shown to be computable in polynomial using Plandowski's result on singleton context-free grammars [20, 11].

5.3 Unitary Theories

Given the identical approach employed in the proofs of Theorem 10 and Theorem 11, it is naturally evident that these results can be generalized to a class of unitary theories that satisfy certain specific conditions. The theory of unary uninterpreted symbols and that of linear arithmetic can then be seen as members of this class. This generalization has been developed in [11]. This generalization is based on defining the concept of a *generic* equation using *context variables*. In the general case, the backward propagation approach requires solving (performing unification on) equations containing context variables.

Unification type of theory of program expressions	Edge Labels	Complexity of assertion checking	Examples	Refs.
Strict Unitary	(a)–(c)	PTIME	`LAE`, `UFS`	[7, 16, 17]
Bitary	(a)–(c)	coNP-hard	`LAE`+`UFS`, `C`	[8]
Finitary,Convex	(a)–(d)	Decidable	`LAE`+`UFS`+`C`+`AC`	[16, 8]
Unitary	(a)-(c),(e)	Decidable	`LAE`,`UUFS`	[11]

Fig. 2. Summary of results. If the program model consists of edges with labels given in Col 2 and the theory underlying the program expressions belongs to the class given in Col 1, then its assertion checking problem has time complexity given in Col 3. Row 1,4 require some additional technical assumptions. Col 4 contains examples of theories for which the corresponding result holds, where `C` denotes commutative functions, and `AC` denotes associative-commutative functions.

6 Forward Abstract Interpretation over Logical Lattices

In the previous sections, we have discussed approaches based on backward propagation on abstract logical lattices. In the case of intraprocedural analysis (Section 4), we were able to perform complete backward propagation on rich logical lattices, but we always required the specification of a goal assertion. In the case of interprocedural analysis (Section 5), we did not need a goal assertion explicitly (since the backward propagation could be done on generic assertions), but we were able to obtain complete procedures for only very simple logical lattices. While these theoretical results are useful in understanding the limits and issues in abstract interpretation over logical lattices, they are of limited help in practice. This is because, in practice, often there is no specification of goal assertions and often the programs use expressions over richer theories. Hence, it is important to consider the problem of *generating* invariants by performing forward abstract interpretation over logical lattices.

We restrict our discussion to an intraprocedural setting in this section. It is evident that building an efficient forward logical abstract interpretation on programs with edge labels (a)-(c) over a logical lattice $(A, \sqsubseteq)$ requires:

R1. Given two elements E_1 and E_2 in A, the join $E_1 \sqcup E_2$ should be efficiently computable,
R2. Given $E \in A$, and a program variable x, the best over-approximation not containing x, that is, $\sqcap\{E' \mid E \sqsubseteq E', E' \text{ does not contain } x\}$, is efficiently computable,
R3. Given $E \in A$, and a ground equation $x = e$ where x does not occur in E, $E \sqcap \{x = e\}$ is efficiently computable,
R4. Given $E, E' \in A$, the relation $E \sqsubseteq E'$ is efficiently decidable.

Requirements R1 and R2 ensure that assertions can be propagated forward, respectively, at join points and across non-deterministic assignments. Requirements R2 and R3 together guarantee that assertions can be propagated forward across assignments. Requirement R4 helps in detecting when a fixpoint is reached.

Finding expressive logical lattices for which the Requirements R1–R4 can be satisfied is one of the challenges in building expressive and scalable abstract interpreters. Since, by Definition 2, $\sqsubseteq$ is generally (some refinement of) the implication relation $\Rightarrow_{\mathbb{T}}$ in a logical theory $\mathbb{T}$, Requirement R4 is often easily fulfilled using existing *decision procedures* for various theories. Requirement R3 is also easy to satisfy for many logical lattices since the domain of a logical lattice is frequently closed under conjunction. Requirement R2 asks for a quantifier elimination procedures, but the result is expected to lie in a restricted subclass A of logical formulas. The problems mentioned in Requirements R1 and R2 are not so well-studied in the theorem proving community. We mention some of the known results here. Karr [14] presented a join algorithm for the linear arithmetic logical lattice. Mine [15] discussed the logical lattice on the octagon abstract domain. Join algorithms for nonlinear polynomial abstract domain were studied by [22], and those for initial term algebra by [7, 23, 13]. We note here that computing join is often more difficult than deciding $\sqsubseteq$ (satisfiability decision procedure) since $E \sqsubseteq E'$ reduces to checking equivalence of $E \sqcap E'$ and E.

The intuitive choice for the domain of a logical lattice is the conjunction of atomic formulas in the theory. The natural choice for the ordering relation $\sqsubseteq$ is the logical implication relation $\Rightarrow_{\mathbb{T}}$ in the theory. Unfortunately, as we saw in Example 6 and Example 7, these common choices, when put together, need not yield a lattice. This problem can often be solved by restricting the domain or the ordering relation.

6.1 Combining Logical Interpreters

One attractive feature of logical lattices is that there is a natural notion of combination of logical lattices that corresponds directly to the notion of combination of logical theories. This notion is called the *logical product* of logical lattices [9]. The logical product is more expressive than the direct product or reduced product [5, 3] of lattices.

A natural question related to logical product of logical lattices is the following: Given abstract interpreters for the individual logical lattices (in the form of, say, witnesses for the satisfiability of the four Requirements R1–R4, can we obtain an abstract interpreter for the logical product?

A positive answer for this question, under certain assumptions on the logical theories underlying the logical lattices, was provided in [9]. This combination result (and the assumptions on the logical theories) are inspired by the Nelson-Oppen combination method for decision procedures [19]. Specifically, we require the individual logical theories to be disjoint, convex, and stably-infinite.

6.2 Richer Logical Lattices

We briefly discuss extensions to the program model to make it more realistic. In the program model discussed above, conditionals were abstracted as non-deterministic choices. In reality, conditionals are important when reasoning about programs. This is, however, easily fixed as forward propagation based

logical interpreters can use the meet operation to handle conditionals. As pointed out above, adding conditionals makes the assertion checking problem undecidable in many cases. Hence, abstract interpretation approaches necessarily lose either completeness or termination on this rich program model.

A second crucial feature absent in the program model of Section 2.1 is the *heap* and assignments that manipulate the heap. Modeling the heap and analyzing the properties of data-structures in the heap is important for verifying real programs. In programs that manipulate the heap, the lvalue *lval* of an assignment, $lval := e$, need not always be a variable. In such a case, one of the first hurdles to overcome is the issue of aliasing. For example, using C notation, an assignment to $x \rightarrow p$ can change the value of $y \rightarrow p$, if x and y contain the same value. As a result, propagating assertions through assignment edges becomes highly nontrivial in presence of aliasing. A second hurdle when analyzing heap manipulating programs is that all interesting invariants of such programs are about unbounded data-structures in heaps. For example, an invariant could state that all elements of an array or list are initialized. Simpler abstract domains like the ones discussed in previous sections are not expressive to represent such invariants.

These two issues can both be resolved using a very carefully designed logical lattice that can express quantified formulas. Quantified formulas can be used to represent aliasing information. Furthermore, it can also be used to represent invariants of unbounded data-structures. Logical interpretation, but over these richer abstract domains, is a promising approach for designing analysis tools for verification of real and complex code.

While designing useful logical abstract domains, one has to make sure that, in an effort to improve expressiveness, the computational aspects outlined as Requirements R1– R4 are not compromised. The reader is referred to [12] for an abstract domain that includes *quantified formulas*, but that is parameterized by a base abstract domain. A logical interpreter over the rich domain is built using a logical interpreter over the simpler base domain. A good choice of base domain gives an expressive and efficient quantified abstract domain [12].

6.3 Interface for a Logical Lattice: The Boolean Interface

In the effort to built new logical interpreters by using existing logical interpreters, we realized that we need a richer interface from an existing logical interpreter. Apart from the ability to (a) compute meet (which is an over-approximation of conjunction) and join (which is an over-approximation of disjunction) on the logical lattice and (b) check for the ordering relation (which is often just a satisfiability checking procedure), we also need functions that (c) compute good over-approximations of join and meet (especially in lattices that are not complete), and (d) good under-approximations of meet (conjunction) and join (disjunction). Additionally, we require the ability to (e) compute good over- and under-approximations of the operation of projecting out a variable (quantifier elimination on a class of formulas that form the domain of the logical lattice). These operations are often required to be done under some context, that is,

under the assumption that certain formulas are known to be true. The exact utility of these interface functions in designing new logical interpreters is beyond the scope of this paper.

7 Conclusion

This paper presents *logical interpretation* - a static analysis approach to checking and generating rich invariants based on using logical lattices. Logical interpretations provide a new paradigm for embedding theorem proving techniques in program analysis tools. The various *satisfiability modulo theory* solvers provide just one of the essential interface functions, and many others are required to build logical interpreters. These other little engines of proof are procedures that perform unification, context unification, matching, and compute over- and under-approximations of conjunction, disjunction, and quantifier elimination on a class of formulas in some theory. The design of effective logical interpreters tries to achieve a balance between *expressiveness* and *computational efficiency*. Well-designed logical abstract domains have the potential of making significant impact on automatically verifying the partial correctness of significant parts of domain-specific software components.

Acknowledgments. The authors thank N. Shankar for helpful feedback.

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 A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 8, pages 445–532. Elsevier Science, 2001.
3. P. Cousot. Forward relational infinitary static analysis, 2005. Lecture Notes, Available at `http://web.mit.edu/afs/athena.mit.edu/course/16/16.399/www`.
4. 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.
5. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *6th ACM Symp. on POPL*, pages 269–282, 1979.
6. C. Flanagan, K. Rustan M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI*, pages 234–245, 2002.
7. S. Gulwani and G. C. Necula. A polynomial-time algorithm for global value numbering. In *Static Analysis Symposium*, volume 3148 of *LNCS*, pages 212–227, 2004.
8. S. Gulwani and A. Tiwari. Assertion checking over combined abstraction of linear arithmetic and uninterpreted functions. In *ESOP*, volume 3924 of *LNCS*, 2006.
9. S. Gulwani and A. Tiwari. Combining abstract interpreters. In *PLDI*, June 2006.
10. S. Gulwani and A. Tiwari. Assertion checking unified. In *Proc. Verification, Model Checking and Abstract Interpretation, VMCAI 2007*, volume 4349 of *LNCS*. Springer, 2007. To appear.

11. S. Gulwani and A. Tiwari. Computing procedure summaries for interprocedural analysis. In *Proc. European Symp. on Programming, ESOP 2007*, LNCS. Springer, 2007. To appear.
12. S. Gulwani and A. Tiwari. Static analysis of heap manipulating low-level software. In *CAV*, LNCS, 2007. To appear.
13. S. Gulwani, A. Tiwari, and G. C. Necula. Join algorithms for the theory of uninterpreted symbols. In *Conf. Found. of Soft. Tech. and Theor. Comp. Sci., FST&TCS '2004*, volume 3328 of *LNCS*, pages 311–323, 2004.
14. M. Karr. Affine relationships among variables of a program. *Acta Informatica*, 6:133–151, 1976.
15. A. Mine. The octagon abstract domain. *Higher Order Symbol. Comput.*, 19(1):31–100, 2006.
16. M. Müller-Olm, O. Rüthing, and H. Seidl. Checking Herbrand equalities and beyond. In *VMCAI*, volume 3385 of *LNCS*, pages 79–96. Springer, January 2005.
17. M. Müller-Olm and H. Seidl. A note on Karr's algorithm. In *31st International Colloquium on Automata, Languages and Programming*, pages 1016–1028, 2004.
18. M. Müller-Olm and H. Seidl. Precise interprocedural analysis through linear algebra. In *31st ACM Symposium on POPL*, pages 330–341, January 2004.
19. G. Nelson and D. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, October 1979.
20. W. Plandowski. Testing equivalence of morphisms on context-free languages. In *Algorithms - ESA '94*, volume 855 of *LNCS*, pages 460–470. Springer, 1994.
21. T. Reps. On the sequential nature of interprocedural program-analysis problems. *Acta Informatica*, 33(8):739–757, November 1996.
22. E. Rodriguez-Carbonell and D. Kapur. An abstract interpretation approach for automatic generation of polynomial invariants. In *11th Static Analysis Symposium*, volume 3148 of *Lecture Notes in Computer Science*. Springer, 2004.
23. 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.
24. N. Shankar. Little engines of proof. In *FME 2002: International Symposium of Formal Methods Europe*, volume 2391 of *LNCS*, pages 1–20. Springer, 2002.
25. 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.