Abstractions from Proofs[*]

Thomas A. Henzinger Ranjit Jhala Rupak Majumdar
EECS Department, University of California
Berkeley, CA 94720-1770, U.S.A.
{tah,jhala,rupak}@eecs.berkeley.edu

Kenneth L. McMillan
Cadence Berkeley Labs.
Berkeley, CA, U.S.A.
mcmillan@cadence.com

Abstract

The success of model checking for large programs depends crucially on the ability to efficiently construct parsimonious abstractions. A predicate abstraction is parsimonious if at each control location, it specifies only relationships between *current* values of variables, and only those which are required for proving correctness. Previous methods for automatically refining predicate abstractions until sufficient precision is obtained do not systematically construct parsimonious abstractions: predicates usually contain symbolic variables, and are added heuristically and often uniformly to many or all control locations at once. We use Craig interpolation to efficiently construct, from a given abstract error trace which cannot be concretized, a parsominous abstraction that removes the trace. At each location of the trace, we infer the relevant predicates as an interpolant between the two formulas that define the past and the future segment of the trace. Each interpolant is a relationship between current values of program variables, and is relevant only at that particular program location. It can be found by a linear scan of the proof of infeasibility of the trace.

We develop our method for programs with arithmetic and pointer expressions, and call-by-value function calls. For function calls, Craig interpolation offers a systematic way of generating relevant predicates that contain only the local variables of the function and the values of the formal parameters when the function was called. We have extended our model checker BLAST with predicate discovery by Craig interpolation, and applied it successfully to C programs with more than 130,000 lines of code, which was not possible with approaches that build less parsimonious abstractions.

Categories and Subject Descriptors: D.2.4 [Software Engineering]: Software/Program Verification; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs.

General Terms: Languages, Verification, Reliability.

Keywords: Software model checking, predicate abstraction, counterexample analysis.

1 Introduction

Increasing dependency on software systems amplifies the need for techniques that can analyze such systems for errors and prove them safe. The two most desirable features of such analyses is that they be *precise* and *scalable*. Precision is required so that the analysis does not report errors where none exist, nor assert correctness when there are bugs. Scalability is necessary so that the method works for large software systems, where the need for analysis is most acute. These two features are often mutually exclusive: flow based interprocedural analyses [11, 15] achieve scalability by fixing a small domain of dataflow facts to be tracked, and compute flow functions over the abstract semantics of the program on this fixed set. For complicated properties, if the set of facts that are tracked is too small, many false positives are reported. Model checking based approaches [25] on the other hand offer the promise of precision as they are path-sensitive, but they often track too many facts, so state explosion comes in the way of scalability.

To avoid the pitfalls arising from using a fixed set of facts, much recent interest has focused on interprocedural analyses that automatically tune the precision of the analysis using false positives, *i.e.,* in a *counterexample-guided* manner. These start with some coarse abstract domain and successively refine the domain by adding facts that make the abstraction sufficiently precise to refute spurious counterexamples [4, 5, 8, 12, 19]. The "facts" are predicates that relate values of programs variables. While this approach holds the promise of precision, there are several obstacles that must be overcome before it can be scaled to very large programs. The first challenge is how to efficiently analyze a false positive and learn from it a *small* set of predicates such that the refined abstraction does not contain the spurious error trace. The second, closely related problem is how to use the discovered predicates parsimoniously. The number of facts that one needs to track grows with the size of the program being analyzed. However, most predicates are only locally useful, *i.e.,* only useful when analyzing certain parts of the program, and irrelevant in others. If locality is not exploited, then the sheer number of facts may render the abstract system too detailed to be amenable to analysis, as the size of the abstract system grows exponentially with the number of predicates.

[*] This research was supported in part by the NSF grants CCR-0085949, CCR-0234690, and ITR-0326577, by the AFOSR grant F49620-00-1-0327, by the DARPA grant F33615-00-C-1693, and by the ONR grant N00014-02-1-0671.

POPL'04, January 14–16, 2004, Venice, Italy.
Copyright 2004 ACM 1-58113-729-X/04/0001 ...$5.00

We solve both problems using the following observation: the *reason* why an abstract trace is infeasible is succinctly encoded in a *proof* that the trace is infeasible, and so the appropriate abstraction can be culled from the proof. The difficulty in extracting the relevant facts from the proof is that the proof uses the entire history of the trace, while our analysis, and hence our facts, should refer at all points of the trace only to relationships between the "current" values of program variables. Inspired by the use of Craig Interpolation for image-computation in [22], we introduce a method by which the proof can be sliced to yield the relevant facts at each point of the trace. Given an abstract trace, we construct a *trace formula* (TF), which is the conjunction of several constraints, one per instruction, such that the TF is satisfiable iff the trace is feasible. If the trace is infeasible, then we use Craig's interpolation theorem [9] to extract, for each point of the trace, the relevant facts from the proof of unsatisfiability of the TF. Given two formulas φ^- and φ^+, whose conjunction is unsatisfiable, the *Craig interpolant* of (φ^-, φ^+) is a formula ψ such that (i) φ^- implies ψ, (ii) $\psi \wedge \varphi^+$ is unsatisfiable, and (iii) ψ contains only symbols common to φ^- and φ^+. If φ^- is the part of the TF that represents a prefix of an infeasible trace, and φ^+ encodes the remainder of the trace, then the Craig interpolant ψ consists of precisely the facts, as relations between current values of the variables, which need to be known at the cut-point of the trace in order to prove infeasibility.

In this paper, we make the following contributions. First, we show how a proof of unsatisfiability of $\varphi^- \wedge \varphi^+$ can be mined to build the interpolant ψ. The method is efficient in that it uses the same theorem proving effort as is needed to produce a proof of unsatisfiability: the interpolant is generated by a linear scan of the proof. Second, we show how to infer from the interpolants, at each cut-point of an infeasible abstract trace, enough facts to rule out the trace. Moreover, the cut-points provide precise information at which program locations the inferred facts are useful, thus enabling a parsimonious use of predicates. The method can be combined with on-the-fly lazy abstraction [19], and presents an improvement: while in pure lazy abstraction, the set of predicates increases monotonically along a trace, the interpolant predicates may change from one control location to the next, *i.e.,* interpolation provides a procedure for deciding when a predicate becomes irrelevant, and therefore obsolete. We show that the method is both sound and complete, in the sense that if an abstract trace is infeasible, then the interpolants always provide sufficient information for proving infeasibility. Moreover, when abstractions are used as certificates for program correctness following the proof-carrying code paradigm [17], then our parsimonious use of predicates yields more compact proofs.

We illustrate the method on an imperative language of arithmetic and pointer expressions with call-by-value function calls. There are two orthogonal sources of complexity in generating interpolants. The first is function calls and scoping. We want the analysis of a function to be polymorphic in all callers, *i.e.,* the inferred predicates should involve only lvalues that are local to the scope of the function. Here, interpolation provides a procedure for systematically discovering predicates that refer only to (i) the local variables of the function and (ii) the values of the formal parameters at the time of the function call. This allows us to keep the subsequent analysis interprocedural [1, 29]. The second issue is the presence of pointers and aliasing. We want to generate predicates that soundly and completely capture the semantics of programs with pointers and memory allocation. As McCarthy's theory of arrays [21] does not offer suitable interpolants, we need to model memory locations individually.

Finally we report on our experiences with this new kind of abstraction refinement. We have implemented the method in BLAST [19]. Owing to the fact that we only track a few predicates at every program location, we have been able to precisely model check programs considerably larger than have been reported before [7, 17], including a driver that consists of 138,000 lines of C code (we found several behaviors that violate the specification). Even though 382 predicates are required in total to show correctness, the reason the analysis scales is that the average number of relevant predicates at each program location is about 8.

2 Overview

Consider the program fragment shown in Figure 1. The property we wish to check is that locking and unlocking alternate, *i.e.,* between any two calls of `lock` there must be a call of `unlock`, and between any two calls of `unlock` there must be a call of `lock`. Suppose that the code not shown does not contain any calls of `lock` or `unlock`.

```
      while(*){
 1:     if (p1) lock ();          assume p1;
        if (p1) unlock ();        lock ();
        ...                       assume ¬ p1;
 2:     if (p2) lock ();          assume p2;
        if (p2) unlock ();        lock ();
        ...
 n:     if (pn) lock ();
        if (pn) unlock ();
      }
```

Figure 1. Program; spurious counterexample.

A static analysis that tracks whether or not the lock is held returns false positives, *i.e.,* error traces that arise from the imprecision of the analysis. One such spurious error trace is shown on the right in Figure 1. The analysis is fooled because it does not track the predicate $p1$ which correlates the first two `if` statements; either both happen or neither happens, and either way the error cannot be reached. We would like to make the analysis more precise so that this spurious counterexample is eliminated, and we would like to keep refining the analysis until we either have a real counterexample, or, as in this case, the program is proved safe.

Various methods can analyze this particular counterexample and learn that the analysis should track the value of $p1$. Similar counterexamples show that all of the predicates $p1, \ldots, pn$ must be tracked, but as a result, the analysis blows up, because it is not clear when we can "merge" states with different predicate values, and without merging there are an exponential number of states.[1] Notice however that in this program, each predicate is only locally useful, *i.e.,* each pi is "live" only at the statements between labels i and (not including) $i+1$. Hence, to make a *precise* analysis *scalable* we need a method that infers both the predicates and *where* they are useful. In our experience, many large software systems have the property that, while the number of relevant predicates grows with the size of the system, each predicate is useful only in a small part of the state space, *i.e.,* the number of predicates that are relevant at any particular program location is small. By exploiting this property one can make a precise analysis scale to large programs. In particular, our algorithm infers the predicates pi and also that pi is useful only between the labels i and $i+1$; outside these labels, we can forget the value of pi. Thus our analysis considers, in this example, only a linear number of distinct states.

[1] For this particular example certain state representation methods such as BDDs would implicitly merge the states.

1:	$x := ctr$;	$\langle x,1\rangle = \langle ctr,0\rangle$	$x = ctr$
2:	$ctr := ctr+1$;	$\langle ctr,1\rangle = \langle ctr,0\rangle + 1$	$x = ctr-1$
3:	$y := ctr$;	$\langle y,2\rangle = \langle ctr,1\rangle$	$x = y-1$
4:	$\texttt{assume}(x = m)$;	$\langle x,1\rangle = \langle m,0\rangle$	$y = m+1$
5:	$\texttt{assume}(y \neq m+1)$;	$\langle y,2\rangle = \langle m,0\rangle + 1$	

Figure 2. Infeasible trace; constraints; predicates.

The problem, then, is (i) to prove that an abstract trace is infeasible, *i.e.,* it does not correspond to a concrete program trace, and (ii) to extract predicates from the proof, together with (iii) information where to use each predicate, such that the refined abstraction no longer contains the infeasible trace. This is not always as simple as in the locking example; consider the infeasible trace shown in Figure 2, where x, y, ctr, and i are program variables, $:=$ denotes an assignment, and `assume` represents an `if` statement.

Preliminary definitions. Suppose that the formula φ (over the program variables) describes a set of program states, namely, the states in which the values of the variables satisfy φ. The *strongest postcondition* [16] of φ w.r.t. an operation `op` is a formula that describes the set of states reachable from some state in φ by performing the operation `op`. For an assignment, $\mathsf{SP}.\varphi.(x := e)$ is $(\exists x'.\varphi[x'/x] \wedge x = e[x'/x])$, and for an `assume` we have, $\mathsf{SP}.\varphi.(\texttt{assume}\ p) = (\varphi \wedge p)$. For example, $(x = ctr)$ describes the set of states where the value of x equals that of ctr, and $\mathsf{SP}.(x = ctr).(ctr := ctr+1) = (\exists ctr'.x = ctr' \wedge ctr = ctr'+1)$. The operator SP is extended to sequences of operations by $\mathsf{SP}.\varphi.(t_1;t_2) = \mathsf{SP}.(\mathsf{SP}.\varphi.t_1).t_2$. A trace t is *feasible* if $\mathsf{SP}.\texttt{true}.t$ is satisfiable. Given a set P of predicates and a formula φ, the *predicate abstraction* of φ w.r.t. P, written $\alpha.P.\varphi$ is the strongest formula $\hat{\varphi}$ (in the implication ordering) such that (i) $\hat{\varphi}$ is a boolean combination of predicates in P, and (ii) φ implies $\hat{\varphi}$. For example, if $P = \{a = 0, b > 0\}$, then $\alpha.P.(a = b + c \wedge b = 2 \wedge c > 0)$ is $\neg(a = 0) \wedge (b > 0)$. The *abstract strongest postcondition* of φ w.r.t. the operation `op` and predicates P is $\mathsf{SP}_{\mathsf{P}}.\varphi.\mathtt{op} = \alpha.P.(\mathsf{SP}.\varphi.\mathtt{op})$. This is extended to traces by $\mathsf{SP}_{\mathsf{P}}.\varphi.(t_1;t_2) = \mathsf{SP}_{\mathsf{P}}.(\mathsf{SP}_{\mathsf{P}}.\varphi.t_1).t_2$. A trace t is *abstractly feasible* w.r.t. P if $\mathsf{SP}_{\mathsf{P}}.\texttt{true}.t$ is satisfiable. The problem is, given an infeasible trace t, find a set P of predicates such that t is abstractly infeasible w.r.t. P.

Symbolic simulation. One way to solve this problem is to symbolically simulate the trace until an inconsistent state is reached; such inconsistencies can be detected by decision procedures [6]. A dependency analysis can be used to compute which events in the trace cause the inconsistency, and this set of events can then be heuristically minimized to obtain a suitable set of predicates [3, 7]. There are two problems with this approach. First, the inconsistency may depend upon "old" values of program variables, *e.g.,* in the trace shown, such an analysis would use facts like x equals "the value of ctr at line 1," and that the "current" value of ctr is one more than the "value at line 1." In general there may be many such old values, and not only must one use heuristics to deduce which ones to keep, a problem complicated by the presence of pointers and procedures, but one must also modify the program appropriately in order to explicitly name these old values. Intuitively, however, since the program itself does not remember "old" values of variables, and yet cannot follow the path, it must be possible to track relationships between "live" values of variables only, and still show infeasibility. Second, this approach yields no information about *where* a predicate is useful.

Example. We now demonstrate our technique on the trace of Figure 2. First, we build a *trace formula* (TF) which is satisfiable iff the trace is feasible. The TF φ is a conjunction of constraints, one per instruction in the trace. In Figure 2, the constraint for each instruction is shown on the right of the instruction. Each term $\langle\cdot,\cdot\rangle$ denotes a special constant which represents the value of some variable at some point in the trace, *e.g.,* $\langle ctr,1\rangle$ represents the value of ctr after the first two instructions. The constraints are essentially the strongest postconditions, where we give new names to variables upon assignment [10, 14]. Thus, for the assignment in line 1, we generate the constraint $\langle x,1\rangle = \langle ctr,0\rangle$, where $\langle x,1\rangle$ is a new name for the value of x after the assignment, and $\langle ctr,0\rangle$ is the name for ctr at that point. Notice that the "latest" name of a variable is used when the variable appears in an expression on the right. Also note that the conjunction φ of all constraints is unsatisfiable.

To compute the set P of relevant predicates, we could simply take all atomic predicates that occur in the constraints, rename the constants to corresponding program variables, create new names ("symbolic variables") for "old" values of a variable *e.g.,* for $\langle ctr,1\rangle = \langle ctr,0\rangle + 1$ create a new name that denotes the value of ctr at the previous instruction, and add these names as new variables to the program. However, such a set P is often too large, and in practice [3, 19] one must use heuristics to minimize the sets of predicates and symbolic variables by using a minimally infeasible subset of the constraints.

Craig interpolation. Given a pair (φ^-, φ^+) of formulas, an *interpolant* for (φ^-, φ^+) is a formula ψ such that (i) φ^- implies ψ, (ii) $\psi \wedge \varphi^+$ is unsatisfiable, and (iii) the variables of ψ are common to both φ^- and φ^+. If $\varphi^- \wedge \varphi^+$ is unsatisfiable, then an interpolant always exists [9], and can be computed from a proof of unsatisfiability of $\varphi^- \wedge \varphi^+$. We present an algorithm for extracting an interpolant from an unsatisfiability proof in Section 3; if $\mathcal{P}$ is a proof of unsatisfiability of $\varphi^- \wedge \varphi^+$, then we write $\textsc{Itp}.(\varphi^-, \varphi^+).(\mathcal{P})$ for the extracted interpolant for (φ^-, φ^+).

In our example, suppose that $\mathcal{P}$ is a proof of unsatisfiability for the TF φ. Now consider the partition of φ into φ_2^-, the conjunction of the first two constraints ($\langle x,1\rangle = \langle ctr,0\rangle \wedge \langle ctr,1\rangle = \langle ctr,0\rangle + 1$), and φ_2^+, the conjunction of the last three constraints ($\langle y,2\rangle = \langle ctr,1\rangle \wedge \langle x,1\rangle = \langle m,0\rangle \wedge \langle y,2\rangle = \langle m,0\rangle + 1$). The symbols common to φ_2^- and φ_2^+ are $\langle x,1\rangle$ and $\langle ctr,1\rangle$; they denote, respectively, the values of x and ctr after the first two operations of the trace. The interpolant $\textsc{Itp}.(\varphi_2^-, \varphi_2^+).(\mathcal{P})$ is $\psi_2 = (\langle x,1\rangle = \langle ctr,1\rangle - 1)$. Let $\hat{\psi}_2$ be the formula obtained from ψ_2 by replacing each constant with the corresponding program variable, *i.e.,* $\hat{\psi}_2 = (x = ctr - 1)$. Since ψ_2 is an interpolant, φ_2^- implies ψ_2, and so $x = ctr - 1$ is an overapproximation of the set of states that are reachable after the first two instructions (as the common constants denote the values of the variables after the first two instructions). Moreover, by virtue of being an interpolant, $\psi_2 \wedge \varphi_2^+$ is unsatisfiable, meaning that from no state satisfying $\hat{\psi}_2$ can one execute the remaining three instructions, *i.e.,* the suffix of the trace is infeasible for all states with $x = ctr - 1$. If we partition the TF φ in this way at each point $i = 1,\ldots,4$ of the trace, then we obtain from $\mathcal{P}$ four interpolants $\psi_i = \textsc{Itp}.(\varphi_i^-, \varphi_i^+).(\mathcal{P})$, where φ_i^- is the conjunction of the first i constraints of ϕ, and φ_i^+ is the conjunction of the remaining constraints. Upon renaming the constants, we arrive at the formulas $\hat{\psi}_i$, which are shown in the rightmost column of Figure 2. We collect the atomic predicates that occur in the formulas $\hat{\psi}_i$, for $i = 1,\ldots,4$, in the set P of predicates.

We can prove that the trace is abstractly infeasible w.r.t. P. Intuitively, for each point $i = 1,\ldots,4$ of the trace, the formula $\hat{\psi}_i$ represents an overapproximation of the states s such that s is reachable after the first i instructions of the trace, and the remaining instructions are infeasible from s. From Equation 1 of Section 3, it follows that $\mathsf{SP}.(\hat{\psi}_i).\mathsf{op}_{i+1}$ implies $\hat{\psi}_{i+1}$, for each i. For example, $\mathsf{SP}.(x = ctr - 1).(y := ctr)$ implies $x = y - 1$. Therefore, by adding all predicates from all ψ_i to P, we have $\mathsf{SP}_{\mathsf{P}}.\mathtt{true}.(\mathsf{op}_1;\ldots;\mathsf{op}_i)$ implies $\hat{\psi}_i$. Note that, as the trace is infeasible, $\hat{\psi}_5 = \psi_5 = \mathtt{false}$. Thus, $\mathsf{SP}_{\mathsf{P}}.\mathtt{true}.(\mathsf{op}_1;\ldots;\mathsf{op}_5)$ implies $\mathtt{false}$, *i.e.*, the trace is abstractly infeasible w.r.t. P.

Locality. The interpolants give us even more information. Consider the naive method of looking at just the TF. The predicates we get from it are such that we must track all of them all the time. If, for example, after the third instruction, we forget that x equals the "old" value of *ctr*, then the subsequent `assume` does not tell us that $y = m + 1$ (dropping the fact about x breaks a long chain of reasoning), thus making the trace abstractly feasible. In this example, heuristic minimization cannot rule out any predicates, so all predicates that occur in the proof of unsatisfiability of the TF must be used at all points in the trace. Using the interpolant method, we show that for infeasible traces of length n, the formula $\mathsf{SP}_{\hat{\psi}_n}.(\ldots(\mathsf{SP}_{\hat{\psi}_1}.\mathtt{true}.\mathsf{op}_1)).\mathsf{op}_n$ is unsatisfiable (see Theorem 1 for a precise statement of this). Thus, at each point i in the trace, we need only to track the predicates in $\hat{\psi}_i$. For example, after executing the first instruction, all we need to know is $x = ctr$, after the second, all we need to know is $x = ctr - 1$, after the third, all we need to know is $x = y - 1$, and so on. This gives us a way to localize predicate usage. Thus, instead of a monolithic set of predicates all of which are relevant at all points of a trace, we can deduce a small set of predicates for each point of the trace.

Function calls. The method described above can be generalized to systematically infer well-scoped predicates for an interprocedural analysis [29]. To obtain predicates that contain only locally visible variables, we cut the TF at each point i in a different way. The first part φ^- of a formula pair consists of the constraints from the instructions between and including i_L and i, where i_L is the first instruction of the call body to which i belongs. The second part φ^+ contains all remaining constraints. It can be shown that interpolants for such pairs (φ^-, φ^+) contain only variables that are in scope at the point i, and are sufficient to rule out the false positive when the subsequent static analysis is done in an interprocedural, polymorphic way [1].

Paper outline. Next, we describe how to extract interpolants from proofs. In Section 4 we describe the syntax and semantics of our language. In Section 5 we show how the predicate inference algorithm works for programs without pointers, and in Section 6 we discuss how pointers can be handled. Finally, in Section 7 we report on our experimental results.

3 Interpolants from Proofs

We now present rules that, given a refutation of a formula $\varphi^- \wedge \varphi^+$ in cnf, derives an interpolant ψ for the pair (φ^-, φ^+). Let *FOL* be the set of formulas in the first-order logic of linear equality. A term in the logic is a linear combination $c_0 + c_1 x_1 + \cdots c_n x_n$, where $x_1, \ldots, x_n$ are individual variables and $c_0, \ldots, c_n$ are integer constants. An atomic predicate is either a propositional variable or an inequality of the form $0 \leq x$, where x is a term. A literal is either an atomic predicate or its negation. A clause is a disjunction of literals. Here we consider formulas in the quantifier-free fragment of *FOL*. A sequent is of the form $\Gamma \vdash \Delta$, where Γ and Δ are sets of formulas. The interpretation of $\Gamma \vdash \Delta$ is that the conjunction of the formulas in Γ entails the disjunction of the formulas in Δ.

$$\textsc{Hyp}\ \frac{}{\Gamma \vdash \phi}\ \phi \in \Gamma$$

$$\textsc{Comb}\ \frac{\Gamma \vdash 0 \leq x \quad \Gamma \vdash 0 \leq y}{\Gamma \vdash 0 \leq c_1 x + c_2 y}\ c_{1,2} > 0$$

$$\textsc{Contra}\ \frac{\{\phi_1, \ldots, \phi_n\} \vdash 0 \leq c}{\Gamma \vdash \neg\phi_1, \ldots, \neg\phi_n}\ c < 0$$

$$\textsc{Res}\ \frac{\Gamma \vdash \{\phi\} \cup \Theta \quad \Gamma \vdash \{\neg\phi\} \cup \Theta'}{\Gamma \vdash \Theta \cup \Theta'}$$

Figure 3. Proof system.

We use a theorem prover that generates refutations for sets of clauses using the sequent proof system of Figure 3. In particular, all boolean reasoning is done by resolution. This system is complete for refutation of clause systems over the rationals. We obtain an incomplete system for the integers by systematically translating the literal $\neg(0 \leq x)$ to $0 \leq -1 - x$, which is valid for the integers.

We will use the notation $\phi \preceq \rho$ to indicate that all variables occurring in ϕ also occur in ρ. An *interpolated sequent* is of the form $(\varphi^-, \varphi^+) \vdash \Delta\ [\psi]$, where φ^- and φ^+ are sets of clauses, Δ is a set of formulas, and ψ is a formula. This encodes the following three facts:

$$(1)\ \varphi^- \vdash \psi, \quad (2)\ \psi, \varphi^+ \vdash \Delta, \text{ and} \quad (3)\ \psi \preceq \varphi^+ \cup \Delta.$$

Note that if $(\varphi^-, \varphi^+) \vdash \perp [\psi]$, then ψ is an interpolant for (φ^-, φ^+). We now give a system of derivation rules for interpolated sequents corresponding to the rules of our proof system. These rules are a distillation of methods found in [20, 28]. They are sound, in the sense that they derive only valid interpolated sequents, and also complete relative to our proof system, in the sense that we can translate the derivation of any sequent $\varphi^- \cup \varphi^+ \vdash \Delta$ into the derivation of an interpolated sequent $(\varphi^-, \varphi^+) \vdash \Delta[\psi]$.

We begin with the rule for introduction of hypotheses. Here, we distinguish two cases, depending on whether the hypothesis is from φ^- or φ^+:

$$\boxed{\textsc{Hyp-A}\ \frac{}{(\varphi^-, \varphi^+) \vdash \phi\ [\phi]}\ \ \phi \in \varphi^-}$$

$$\boxed{\textsc{Hyp-B}\ \frac{}{(\varphi^-, \varphi^+) \vdash \phi\ [\top]}\ \ \phi \notin \varphi^-}$$

We take $\top$ here to be an abbreviation for $0 \leq 0$. The rule for inequalities is as follows:

$$\boxed{\textsc{Comb}\ \frac{\begin{array}{c}(\varphi^-, \varphi^+) \vdash 0 \leq x\ [0 \leq x'] \\ (\varphi^-, \varphi^+) \vdash 0 \leq y\ [0 \leq y']\end{array}}{(\varphi^-, \varphi^+) \vdash 0 \leq c_1 x + c_2 y\ [0 \leq c_1 x' + c_2 y']}\ c_{1,2} > 0}$$

In effect, to obtain the interpolant for a linear combination of inequalities from φ^- and φ^+ we just replace the inequalities from φ^+ with $0 \leq 0$. Interpolated sequents derived using these rules satisfy the following invariant.

INVARIANT 1. *For any interpolated sequent of the form* $(\varphi^-, \varphi^+) \vdash 0 \leq x\ [0 \leq x']$, *we have* $\varphi^+ \vdash 0 \leq y'$ *such that* $x = x' + y'$. *Further, for all individual variables* v *such that* $v \not\preceq \varphi^+$, *the coefficients of* v *in* x *and* x' *are equal.*

Using this invariant, we can show that the above rules to generate interpolants are sound.

As an example, Figure 4 shows the derivation of an interpolant for the case where φ^- is $(0 \leq y - x)(0 \leq z - y)$ and φ^+ is $(0 \leq x - z - 1)$. In the figure, we abbreviate $(\varphi^-, \varphi^+) \vdash \phi\ [\psi]$ to $\vdash \phi\ [\psi]$. Using the above rules, we derive the sequent $\vdash 0 \leq -1\ [0 \leq z - x]$. Since $0 \leq -1$ is equivalent to $\perp$, it follows that $0 \leq z - x$ is an interpolant for (φ^-, φ^+) (which the reader may wish to confirm).

Inequality reasoning is connected to Boolean reasoning in our system via the CONTRA rule. The corresponding interpolation rule is as follows:

$$\boxed{\textsc{Contra}\ \frac{(\{a_1, \ldots, a_k\}, \{b_1, \ldots, b_m\}) \vdash \perp\ [\psi]}{\begin{array}{c}(\varphi^-, \varphi^+) \vdash \neg a_1, \ldots, \neg a_k, \neg b_1, \ldots, \neg b_m \\ [(\neg a_1 \vee \cdots \vee \neg a_k) \vee \psi]\end{array}}}$$

This rule is sound because both the consequent and the interpolant it generates are tautologies. Moreover, we apply the side condition that all the b_i are literals occurring in φ^+, while all the a_i are literals not occurring in φ^+. This establishes the following invariant.

INVARIANT 2. *For any interpolated sequent* $(\varphi^-, \varphi^+) \vdash \Theta\ [\psi]$, *the set* Θ *is a collection of literals, and* ψ *is of the form* $\phi \vee \rho$, *where* ϕ *is the disjunction of those literals in* Θ *not occurring in* φ^+.

Notice that, provided φ^- is in clause form, our two hypothesis introduction rules HYP-A and HYP-B also establish this invariant (any clause from φ^- can be rewritten into the form $\phi \vee \rho$ required by the invariant).

Now, we introduce two interpolation rules for resolution: one for resolution on a literal occurring in φ^+, and the other for resolution on a literal not occurring in φ^+:

$$\boxed{\textsc{Res-A}\ \frac{\begin{array}{c}(\varphi^-, \varphi^+) \vdash \phi, \Theta\ [(\phi \vee \rho) \vee \psi] \\ (\varphi^-, \varphi^+) \vdash \neg\phi, \Theta'\ [(\neg\phi \vee \rho') \vee \psi']\end{array}}{(\varphi^-, \varphi^+) \vdash \Theta, \Theta'\ [(\rho \vee \rho') \vee (\psi \vee \psi')]}}$$

$$\boxed{\textsc{Res-B}\ \frac{(\varphi^-, \varphi^+) \vdash \phi, \Theta\ [\rho \vee \psi] \qquad (\varphi^-, \varphi^+) \vdash \neg\phi, \Theta'\ [\rho' \vee \psi']}{(\varphi^-, \varphi^+) \vdash \Theta, \Theta'\ [(\rho \vee \rho') \vee (\psi \wedge \psi')]}}$$

In effect, when resolving a literal on the φ^- side (not occurring in φ^+) we take the disjunction of the interpolants, and when resolving a literal on the φ^+ side (occurring in φ^+) we take the conjunction of the interpolants. Using Invariant 2 we can show that these rules are sound.

As an example, Figure 5 shows a derivation of an interpolant for (φ^-, φ^+), where φ^- is $(b)(\neg b \vee c)$ and φ^+ is $(\neg c)$. Using the resolution rule, we derive the sequent $(\varphi^-, \varphi^+) \vdash \perp\ [c]$. Thus c is an interpolant for (φ^-, φ^+).

Using the invariants given above, we can also show that for every derivation $\mathcal{P}$ of a sequent $(\varphi^-, \varphi^+) \vdash \phi$ in our original proof system, there is a corresponding derivation $\mathcal{P}'$ of an interpolated sequent of the form $(\varphi^-, \varphi^+) \vdash \phi\ [\psi]$. We will refer to the interpolant ψ thus derived as $\textsc{Itp}.(\varphi^-, \varphi^+).(\mathcal{P})$. Using the same proof but partitioning the antecedent differently, we can obtain related interpolants. For example, we can show the following fact:

$$\textsc{Itp}.(\varphi^-, \phi \cup \varphi^+).(\mathcal{P}) \wedge \phi \implies \textsc{Itp}.(\varphi^- \cup \phi, \varphi^+).(\mathcal{P}) \qquad (1)$$

This fact will be useful later in showing that a set of interpolants derived from an infeasible program trace provides a sufficient set of predicates to rule out that trace.

We can also give interpolation rules for treating equalities and uninterpreted functions. This is omitted here due to space considerations. We also note that some useful theories do not have the Craig interpolation property. For example, interpolants do not always exist in the quantifier-free theory of arrays (with `sel` and `upd` operators) [21]. For this reason, we avoid arrays in this work, although the use of array operators would simplify the theory somewhat.

4 Languages and Abstractions

We illustrate our algorithm on a small imperative language with integer variables, references, and functions with call-by-value parameter passing.

Syntax. We consider a language with integer variables and pointers. Lvalues (memory locations) in the language are declared variables or dereferences of pointer-typed expressions. We assume for simplicity that at the beginning of a function, memory is allocated for each reference variable. Arithmetic comparison or pointer equality constitute boolean expressions. For any lvalue l, let $\mathsf{typ}.l$ be the type of l; $\mathsf{typ}.x$ is the declared type of the variable x in the current scope, and $\mathsf{typ}.*l_1$ is τ if $\mathsf{typ}.l_1$ is $\mathtt{ref}\ \tau$ (and there is a type error otherwise). The operation $l := e$ writes the value of the expression e in the memory location l; the operation $\mathtt{assume}(p)$ succeeds if the boolean expression p evaluates to **`true`**, the program halts otherwise. An operation $f(x_1, \ldots, x_n)$ corresponds to a call to function f with actual parameters x_1 to x_n, and **`return`** corresponds to a return to the caller. We assume that all operations are type safe.

We represent each function f as a *control flow automaton (CFA)* $C_f = (L_f, E_f, l_f^0, Op_f, V_f)$. The CFA C_f is a rooted, directed graph with a set of vertices $L_f \subseteq PC$ which correspond to program locations, a set of edges $E_f \subseteq L_f \times L_f$, a special start location $l_f^0 \in L_f$, a labeling function $Op_f \colon \mathrm{E}_f \to Ops$ that yields the operation labeling each edge, and a set of typed local variables $V_f \subseteq Lvals$. The set V_f of local variables has a subset $X_f \subseteq V_f$ of formal parameters passed to f on a call, and a variable $r_f \in V_f$ that stores the return value. A *program* is a set of CFAs $\mathbb{P} = \{C_{f_0}, \ldots, C_{f_k}\}$, where each C_{f_i} is the CFA for a function f_i. There is a special function **`main`** and corresponding CFA $C_{\mathtt{main}}$, program execution begins there.

Let $PC = \cup\{L_f \mid C_f \in \mathbb{P}\}$ be the set of program locations. A *command* is a pair $(\mathsf{op}, pc) \in Ops \times PC$. A *trace* of $\mathbb{P}$ is a sequence of commands $(\mathsf{op}_1 : pc_1); \ldots; (\mathsf{op}_n : pc_n)$, where (1) there is an edge $(l_{\mathtt{main}}^0, pc_1)$ in $C_{\mathtt{main}}$ such that $Op(l_{\mathtt{main}}^0, pc_1) = \mathsf{op}_1$, (2) if op_i is a function call $f(\cdots)$, then pc_i is l_f^0, the initial location of function f, (3) the function calls and returns are properly matched, so if op_i is a **`return`**, then pc_i is the control location immediately after the call in the appropriate caller, (4) otherwise (if op_i is not a function call or return), there is an edge $(pc_{i-1}, pc_i) \in E_f$ such that $Op(pc_{i-1}, pc_i) = \mathsf{op}_i$ (where f is the CFA such that $pc_i \in L_f$). For a trace $t = (\mathsf{op}_1 : pc_1); \ldots; (\mathsf{op}_n : pc_n)$, let $Cl.t$ be a function such that if op_i is a function call, then $\mathsf{op}_{Cl.t.i}$ is the matching return, and $Cl.t.i = n$ otherwise. For each $1 \leq i \leq n$, define $L.t.i$ to be $\max\{j \mid\ j \leq i$, and op_j is a function call, and $Cl.t.j \geq i\}$, and 0 if this set is empty, and $R.t.i = Cl.t.(L.t.i)$. For a trace $t = (\mathsf{op}_1 : pc_1); \ldots; (\mathsf{op}_n : pc_n)$, and $1 \leq i \leq n$, the position $L.t.i$ has the call that begins the scope to which op_i belongs, and $R.t.i$ has the return that ends that scope. For simplicity of notation, we assume that every function call returns.

$$\textsc{Comb}\ \dfrac{\textsc{Comb}\ \dfrac{\textsc{Hyp-A}\ \dfrac{}{\vdash 0 \leq y-x\ [0 \leq y-x]} \qquad \textsc{Hyp-A}\ \dfrac{}{\vdash 0 \leq z-y\ [0 \leq z-y]}}{\vdash 0 \leq z-x\ [0 \leq z-x]} \qquad \textsc{Hyp-B}\ \dfrac{}{\vdash 0 \leq x-z-1\ [0 \leq 0]}}{\vdash 0 \leq -1\ [0 \leq z-x]}$$

Figure 4. Deriving an interpolant.

$$\textsc{Res}\ \dfrac{\textsc{Res}\ \dfrac{\textsc{Hyp-A}\ \dfrac{}{\vdash b\ [b \vee \bot]} \qquad \textsc{Hyp-A}\ \dfrac{}{\vdash \neg b, c\ [\neg b \vee c]}}{\vdash c\ [\bot \vee (\bot \vee c)]} \qquad \textsc{Hyp-B}\ \dfrac{}{\vdash \neg c\ [\bot \vee \top]}}{\vdash \bot\ \ [\bot \vee ((\bot \vee c) \wedge \top)]}$$

Figure 5. Deriving an interpolant using resolution.

We fix the following notation for the sequel. We use t for the trace $(\mathsf{op}_1 : pc_1); \ldots ; (\mathsf{op}_n : pc_n)$. For formulas $\phi, \phi_1, \phi_2 \in FOL$, we write $\mathsf{ite}.\phi.\phi_1.\phi_2$ to abbreviate $(\phi \wedge \phi_1) \vee (\neg\phi \wedge \phi_2)$.

Semantics. The semantics of a program is defined over the set v of states. The state of a program contains valuations to all lvalues, the value of the program counter, as well as the call stack. For our purposes, we only consider the *data state*, which is a type-preserving function from all lvalues to values. A *region* is a set of data states. We represent regions using first-order formulas with free variables from the set of program variables. Each operation $\mathsf{op} \in Ops$ defines a state transition relation $\xrightarrow{\mathsf{op}} \subseteq v \times v$ in a standard way [23]. The semantics of a trace can also be given in terms of the *strongest postcondition* operator [16]. Let φ be a formula in FOL representing a region. The strongest postcondition of φ w.r.t. an operation op, written $\mathsf{SP}.\varphi.\mathsf{op}$ is the set of states reachable from states in φ after executing op. The strongest postcondition operator for our language can be computed syntactically as a predicate transformer.

The strongest postcondition operator gives the concrete semantics of the program. Our analyses will consider abstract semantics of programs. The abstract domain will be defined by a set of predicates over the program variables. As we use decision procedures to compute predicate abstractions, we require quantifier-free predicates. For a formula $\varphi \in FOL$ and a set of atomic predicates $P \subseteq FOL$, the *predicate abstraction* of φ w.r.t. the set P is the strongest formula ψ (in the implication order) with atomic predicates from P such that φ implies ψ. Let Π: $PC \to 2^{FOL}$ be a mapping from program locations to sets of atomic predicates. The operator SP_Π is the abstraction of the operator SP w.r.t. Π. Formally, let φ denote a set of states, and let $(\mathsf{op} : pc)$ be a command. Then $\mathsf{SP}_\Pi.\varphi.(\mathsf{op} : pc)$ is the predicate abstraction of $\mathsf{SP}.\varphi.\mathsf{op}$ w.r.t. $\Pi.pc$.

Let SP be a syntactic strongest postcondition operation, and SP_Π its abstraction w.r.t. the mapping Π. For any trace t, the trace t is (1) *feasible* if there exist states $s_0, s_1, \ldots, s_n \in v$ such that $s_j \xrightarrow{\mathsf{op}_j} s_{j+1}$ for $j = 0, \ldots, n-1$, and infeasible otherwise; (2) SP-*feasible* if $\mathsf{SP}.\mathtt{true}.t$ is satisfiable, and SP-*infeasible* otherwise; and (3) Π-*feasible* if $\mathsf{SP}_\Pi.\mathtt{true}.t$ is satisfiable, and Π-*infeasible* otherwise. The two notions (1) and (2) coincide [23].

Subclasses of programs. A program is *flat* if it is a singleton $\{C_{\mathtt{main}}\}$, and there is no edge (pc, pc') in $C_{\mathtt{main}}$ such that $Op(pc, pc')$ is a function call or a return. A program is *pointer-free* if all lvalues and expressions have type $\mathtt{int}$. Specifically, a pointer-free program does not have any references. In the following, we shall consider four classes of programs: (Class PI) flat and pointer-free, (Class PII) pointer-free (but not flat), (Class PIII) flat (but not pointer-free), and (Class PIV) the class of all programs. For each class, we define a syntactic predicate transformer SP that takes a formula $\varphi \in FOL$ and an operation $\mathsf{op} \in Ops$ and returns the strongest postcondition $\mathsf{SP}.\varphi.\mathsf{op}$. We also define the predicate abstraction SP_Π of SP. Finally, we present an algorithm $\mathsf{Extract}$ that takes a trace t and returns a mapping Π from PC to sets of atomic predicates in FOL. The following theorem relates the different notions of feasibility.

THEOREM 1. *Let t be a trace of a program P of class* PI, PII, PIII, *or* PIV. *The following are equivalent:*

1. *t is infeasible (or equivalently, t is* SP*-infeasible).*

2. *t is* SP_Π*-infeasible for* $\Pi = \mathsf{Extract}.t$.

In particular, Theorem 1 states that our predicate discovery procedure $\mathsf{Extract}$ is *complete* for each class: for an infeasible trace t, the predicate map $\mathsf{Extract}.t$ is precise enough to make the trace SP_Π-infeasible (*i.e.,* the infeasible trace t is not a trace of the abstraction). If all integer variables are initialized to some default integer value, say 0, then all satisfying assignments of the SP of a trace will be integral even if the SP is interpreted over the rationals. Thus, if the trace is infeasible, our proof system can derive the unsatisfiability of the strongest postcondition.

In the next two sections, we describe in detail how we mine predicates from proofs of unsatisfiability of spurious error traces. First we consider programs in the classes PI and PII. We then generalize our results to the classes PIII and PIV. For a given trace t of each class of program, we define the following operators. First, we define the concrete and abstract strongest postcondition operators SP and SP_Π, which take a formula and an operation and return a formula, and we extend them to the entire trace t. Next, we define an operator Con, which returns a *constraint map*. This is a function that maps each point i of the trace t to a *constraint* that corresponds to the ith operation of the trace. The conjunction of the constraints that are generated at all points of t is the trace formula (TF) for t, which is satisfiable iff the trace is feasible. Finally, we define the procedure $\mathsf{Extract}$, which uses a proof of unsatisfiability of the TF to compute a function Π that maps each program location to a set of atomic predicates such that the trace t is Π-infeasible.

t	$\mathsf{SP}.\varphi.t$	$\mathsf{SP}_{\Pi}.\varphi.t$	$\mathsf{Con}.(\theta,\Gamma).t$
$(x := e : pc_i)$	$\exists x'.\ (\varphi[x'/x] \land\ x = e[x'/x])$ where x' is a fresh variable	$\alpha.(\Pi.pc).(\mathsf{SP}.\varphi.t)$	$(\theta',\Gamma[i \mapsto (\mathsf{Sub}.\theta'.x = \mathsf{Sub}.\theta.e)])$ where $\theta' = \mathsf{Upd}.\theta.\{x\}$
$(\mathtt{assume}(p) : pc_i)$	$\varphi \land\ p$	”	$(\theta,\Gamma[i \mapsto \mathsf{Sub}.\theta.p])$
$t_1;t_2$	$\mathsf{SP}.(\mathsf{SP}.\varphi.t_1).t_2$	$\mathsf{SP}_{\Pi}.(\mathsf{SP}_{\Pi}.\varphi.t_1).t_2$	$\mathsf{Con}.(\mathsf{Con}.(\theta,\Gamma).t_1).t_2$
$(y := f(\vec{e}) : pc_i);$ $t_1;$ $(\mathtt{return} : pc_j)$	$\exists y',\vec{\phi}.\ \varphi[y'/y]$ $\land\ \exists r.\ \vec{\phi} = \vec{e}[y'/y]$ $\land\ \exists V_f^-.\ \mathsf{SP}.\mathtt{true}.t_1$ $\land\ y = r$ where y' is fresh $\vec{x}$ are the formals of f r is the return variable of f $V_f^- = V_f \setminus (sym_f \cup \{r\})$	$\alpha.(\Pi.pc_j).$ $(\exists y',\vec{\phi}.\ \varphi[y'/y]$ $\land\ \exists r.\ \vec{\phi} = \vec{e}[y'/y]$ $\land\ \exists V_f^-.\ \mathsf{SP}_{\Pi}.\mathtt{true}.t_1$ $\land\ y = r)$	(θ',Γ') where $\theta' = \mathsf{Upd}.\theta.\{y\}$ $\theta_I = \mathsf{Upd}.\theta'.V_f$ $\Gamma_I = \Gamma[i \mapsto \mathsf{Sub}.\theta_I.\vec{\phi} = \mathsf{Sub}.\theta.\vec{e}]$ $(\theta_O,\Gamma_O) = \mathsf{Con}.(\theta_I,\Gamma_I).t_1$ $\Gamma' = \Gamma_O[j \mapsto \mathsf{Sub}.\theta'.y = \mathsf{Sub}.\theta_O.r]$ r is the return variable of f

Figure 6. Postconditions and constraints for PI and PII traces.

5 Programs without Pointers

5.1 Flat Pointer-free Programs: PI

Strongest postconditions and constraints. We first define the semantics of flat pointer-free programs in terms of a syntactic strongest postcondition operator SP and its predicate abstraction SP_{Π} (w.r.t. a predicate map Π). In Figure 6 the first three rows define the operators SP and SP_{Π} for traces of PI. Each operator takes a formula in *FOL* and returns a formula in *FOL*. The operator SP_{Π} is parameterized by a map Π from *PC* to sets of atomic predicates. With this definition of SP, we can show that Theorem 1 holds.

An *lvalue map* is a function θ from *Lvals* to $\mathbb{N}$. The operator Upd: $(Lvals \to \mathbb{N}) \to 2^{Lvals} \to (Lvals \to \mathbb{N})$ takes a map θ and a set of lvalues L, and returns a map θ' such that $\theta'.l = \theta.l$ if $l \notin L$, and $\theta'.l = i_l$ for a fresh integer i_l if $l \in L$. The function Sub takes an lvalue map θ and an lvalue l and returns $\langle l,\theta.l\rangle$. The function $\mathsf{Sub}.\theta$ is extended naturally to expressions and formulas. A *new* lvalue map is one whose range is disjoint from all other maps. We use lvalue maps to generate trace formulas (TF); at a point in the trace, if the map is θ, then the the pair $\langle l,\theta.l\rangle$ is a special constant that equals the value of l at that point in the trace. Whenever some lvalue l is updated, we update the map so that a fresh constant is used to denote the new value of l. For every such constant $c = \langle l,i\rangle$, let $\mathsf{Clean}.c = l$. The operator Clean can be naturally extended to expressions and formulas of *FOL*.

The constraints are generated by the function Con, which takes a pair (θ,Γ) consisting of an lvalue map θ and a constraint map Γ: $\mathbb{N} \to FOL$, and a command $(pc : \mathsf{op}) \in Cmd$, and returns a pair (θ',Γ') consisting of a new lvalue map and constraint map. We generate one constraint per command. For a trace $t = (\mathsf{op}_1 : pc_1);\ldots;(\mathsf{op}_n : pc_n)$, if $(\theta',\Gamma') = \mathsf{Con}.(\theta,\Gamma).t$ for some initial Γ,θ, then $\Gamma.i$ is the constraint for op_i, and it can be shown by induction on the length of the trace, that the TF $\bigwedge_{1\le i\le n}\Gamma.i$ is satisfiable iff $\mathsf{SP}.\mathtt{true}.t$ is satisfiable. The generated constraints are a skolemized version of the strongest postcondition. The function Con is defined in the first three rows of Figure 6. If the ith operation in the trace is the assignment $x := e$, we first update the map so that a new constant denotes the value of x, and then we have the ith constraint specify that the new constant for x has the same value as the expression e (with appropriate constants plugged in). For an assume operation $\mathtt{assume}(p)$, the constraint stipulates that the constants at that point satisfy the formula p. The constants enable us to encode the entire history of the trace in the constraint map. The following proposition states the correctness of constraint generation.

Algorithm 1 Extract

```
**Input:** an infeasible trace $t = (\mathsf{op}_1 : pc_1);\ldots;(\mathsf{op}_n : pc_n)$.
**Output:** a map $\Pi$ from the locations of $t$ to sets of atomic predicates.
$\Pi.pc_i := \emptyset$ for $1 \le i \le n$
$(\cdot,\Gamma) := \mathsf{Con}.(\theta_0,\Gamma_0)$
$\mathcal{P}$ := derivation of $\bigwedge_{1\le i\le n}\Gamma.i \vdash \mathtt{false}$
for $i := 1$ to $n$ do
  $\varphi^- := \bigwedge_{1\le j\le i}\Gamma.j$
  $\varphi^+ := \bigwedge_{i+1\le j\le n}\Gamma.j$
  $\psi := \textsc{Itp}.(\varphi^-,\varphi^+).(\mathcal{P})$
  $\Pi.pc_i := \Pi.pc_i \cup \mathsf{Atoms}.(\mathsf{Clean}.\psi)$
return $\Pi$.
```

$(\mathtt{assume}(b > 0) : pc_1);$	$\langle b,0\rangle > 0$	
$(c := 2*b : pc_2);$	$\langle c,1\rangle = 2*\langle b,0\rangle$	
$(a := b : pc_3);$	$\langle a,2\rangle = \langle b,0\rangle$	
$(a := a-1 : pc_4);$	$\langle a,3\rangle = \langle a,2\rangle - 1$	φ^-
$(\mathtt{assume}(a < b) : pc_5);$	$\langle a,3\rangle < \langle b,0\rangle$	φ^+
$(\mathtt{assume}(a = c) : pc_6)$	$\langle a,3\rangle = \langle c,1\rangle$	

Figure 7. Cutting a PI trace.

PROPOSITION 1. [Equisatisfiability] *For a trace t let $(\theta,\Gamma) = \mathsf{Con}.(\theta_0,\Gamma_0).t$ and let $\varphi = \bigwedge_{1\le i\le n}\Gamma.i$. The trace t is feasible iff the φ is satisfiable. Moreover, the size of φ is linear in the size of t.*

Predicates from cuts. Given an infeasible trace t, we want to learn a set of predicates that exhibit the infeasibility of the trace. Our method has been described in Section 2 and is made precise in Algorithm 1. Algorithm 1 first sets the map Π to be the empty map. It then generates the constraint map Γ for the entire trace and constructs the TF by conjoining all constraints in the range of Γ. Let $\mathcal{P}$ be a proof of unsatisfiability of the TF. Then for each point i in the trace, we *cut* the constraints into those from the first i commands (φ^-) and those from the remaining commands (φ^+). Using the proof $\mathcal{P}$ we compute the interpolant ψ for (φ^-,φ^+) and add the atomic predicates that occur in ψ after cleaning to the predicate map for pc_i. The correctness of this procedure is stated in Theorem 1.

EXAMPLE 1: Consider the infeasible trace from [3] shown on the left in Figure 7. On the right, the figure shows the result of $\mathsf{Con}.(\theta_0,\Gamma_0).t$, where the initial lvalue map θ_0 maps a, b, and c to 0. To the right of each command is the corresponding con-

$$\begin{array}{ll}
(a := 0 : pc_1); & \langle a,1\rangle = 0 \\
(b := inc(a) : pc_2); & \langle \phi_x,2\rangle = \langle a,1\rangle \\
\quad (\texttt{assume}\ (x = \phi_x) : pc_2); & \quad \langle x,3\rangle = \langle \phi_x,2\rangle \\
\quad (x := x+1 : pc_3); & \quad \langle x,4\rangle = \langle x,3\rangle + 1 \\
\quad (r := x : pc_4); & \quad \langle r,5\rangle = \langle x,4\rangle \\
(\texttt{return}\ r : pc_5); & \langle b,6\rangle = \langle r,5\rangle \\
(\texttt{assume}(a \neq b-1) : pc_6) & \langle a,1\rangle \neq \langle b,6\rangle - 1
\end{array}$$

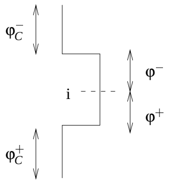

Figure 8. Cutting a PII trace.

straint. When we cut the trace at the fourth location, the resulting pair (φ^-, φ^+) consists of the conjunctions of the constraints from above and below the line, respectively. The interpolant in this case is $\langle a,3\rangle \leq \langle c,1\rangle - 2$, which upon cleaning yields the predicate $a \leq c-2$. Notice that the constants common to both sides of the cut denote the values of the respective variables after the first four operations, and φ^- implies the interpolant. □

5.2 Pointer-free Programs: PII

We now move on to programs with function calls, but no pointers. We assume that there are no global variables, and that each function returns a single integer. When dealing with such programs, the analysis of the previous section may learn predicates that are not well-scoped, *i.e.*, for some location, we may extract predicates that contain variables which are out-of-scope at that location, and such an analysis is not modular. For a modular interprocedural analysis, we need to compute the effect of a function in terms of the arguments passed to it, *i.e.*, we wish to relate the "output" of the function to its "input." The way to do this is to introduce symbolic variables that denote the values the function is called with, perform the analysis using the symbolic variables, and then, at the call site, plug in the actual values for the constants to see the result of the function. This method is known as polymorphic predicate abstraction [1]. We assume that the program has been preprocessed so that (1) for every function and every formal parameter x, the function has a new local, so-called *symbolic* variable ϕ_x, which holds the value of the argument when the function is called and is never written to (the set of symbolic variables of f is sym_f), and (2) the first statement of every function is $\texttt{assume}(\wedge_i x_i = \phi_{x_i})$, where the conjunction ranges over the set X_f of formal parameters. Finally, to avoid clutter in the definitions due to renaming, we assume that different functions have different names for their variables, and that no function calls itself directly (a function f can of course call another function which, in turn, calls f).

Strongest postconditions and constraints. Figure 6 shows the syntactic strongest postcondition operator SP and its predicate abstraction SP_Π for PII traces. The fourth row shows the case of calls and returns (we process the entire call-return subtrace at once). The strongest postcondition for this case is computed as follows: we replace y as for an assignment, we set the symbolic variables ϕ's to the arguments $\vec{e}[y'/y]$, we then compute SP for the body of the function t_1 w.r.t. $\texttt{true}$ (the first operation in the function will equate the formal parameters $\vec{x}$ with the ϕ's), and we subsequently quantify out all locals except the ϕ's, which leaves us with a formula that relates the return variable r to the ϕ's. We then set y to r and quantify out the ϕ's and r. The abstract postcondition does the same, only it uses the the abstract postcondition SP_Π of the function body instead of SP, and then abstracts the resulting formulas using the predicates of pc_j, the location of the caller after the function returns. The generated constraints are again a skolemized version of the strongest postcondition. The constraint for the call command is the formula that equates the ϕ's with the corresponding actual parameters, and the constraint for the return is the formula that equates y with the return value r of the function. To deal with possibly recursive functions, we use a different lvalue map for the constraints of the function body, because we do not want assignments to local variables of the called function to change the values of variables in the calling context. With these definitions, the analogue of Proposition 1 holds for PII programs.

EXAMPLE 2: Consider the trace in Figure 8. The function *inc* has the formal parameter x and return variable r, and returns a value one greater than its input. Assume that we start with an lvalue map θ_0, which maps a, b, x, and ϕ_x to 0. The constraints $\mathsf{Con}.(\theta_0, \texttt{true}).t$ are shown on the right in Figure 8. □

Predicates from cuts. To get well-scoped predicates, we need only to generalize the notion of cuts. For each location i of a trace, instead of partitioning the constraints into those due to commands before i and those originating from commands after i, we use the partition shown in Figure 8. Let i_L (resp., i_R) be the first (resp., last) command in the call body to which the ith command belongs. So i_L is the first command after a function call, and the operation after i_R is a return. We consider four sets of constraints: (1) φ^- corresponds to the commands between (and including) i_L and i, which may include commands that call into and return from other functions, (2) φ^+ corresponds to the commands from $i+1$ to i_R, which may include commands that call into and return from other functions between $i+1$ and i_R, (3) φ_C^- corresponds to the commands in the calling context of command i which occur before the call of the function to which i belongs, and which include the call of the function, and (4) φ_C^+ corresponds to the commands in the calling context which occur after the return of the function to which i belongs, and which include the return. We then construct the interpolant of $(\varphi^-, \varphi^+ \wedge \varphi_C^- \wedge \varphi_C^+)$. One can check that the constants common to φ^- and $\varphi^+ \wedge \varphi_C^+$ are denote the values of locals (including the return variable) at location i, and that the constants common to φ^- and φ_C^- are denote the values of symbolic variables upon entry to the function, which are never changed by the function, and hence also the values of the symbolic variables at location i; (these are also locals of the called function). Hence the interpolant, and thus the predicates we compute, are in terms of variables that are in scope at location i, and they refer only to current values of those variables at location i.

To see why such an interpolant suffices, consider first the partition $(\varphi^- \wedge \varphi^+, \varphi_C^- \wedge \varphi_C^+)$, namely, into constraints that belong to the function body and constraints that belong to the calling context. The resulting interpolant ψ contains the symbolic variables of the function to which the cut-point belongs, as well as the return variable r, *i.e.*, the inputs and output of the called function. Moreover, ψ abstractly summarizes the information about the function call which renders the trace infeasible, because $\psi \wedge (\varphi_C^- \wedge \varphi^+)$ is unsatisfiable. Now, at each point inside the function, we need to know what information is required to show that ψ holds at the end. To get this information, we could compute the interpolant of $(\varphi^-, \varphi^+ \wedge \neg\psi)$, but since $\neg\psi$ is implied by $\varphi_C^- \wedge \varphi_C^+$, we can instead directly compute the interpolant of $(\varphi^-, \varphi^+ \wedge (\varphi_C^- \wedge \varphi_C^+))$. The resulting predicate discovery algorithm Extract for traces of PII programs is shown in Figure 2. The correctness of this procedure is stated in Theorem 1.

EXAMPLE 3: Recall the trace and constraints from Figure 8. The formulas that result from cutting the trace at the fourth location are $\varphi^- = (\Gamma.3 \wedge \Gamma.4) = (\langle x,3\rangle = \langle \phi_x,2\rangle \wedge \langle x,4\rangle = \langle x,3\rangle + 1)$ and

Algorithm 2 Extract

```
Input: an infeasible trace $t = (\mathtt{op}_1 : pc_1);\ldots;(\mathtt{op}_n : pc_n)$.
Output: a map $\Pi$ from the locations of $t$ to sets of atomic predicates.
$\Pi.pc_i := \emptyset$ for $1 \le i \le n$
$(\cdot,\Gamma) := \mathsf{Con}.(\theta_0,\Gamma_0)$
$\mathcal{P}$ := derivation of $\bigwedge_{1\le i\le n}\Gamma.i \vdash \mathtt{false}$
for $i := 1$ to $n$ do
    $(i_L, i_R) := (L.t.i, R.t.i)$
    $\varphi^- := \bigwedge_{i_L+1\le j\le i}\Gamma.j$
    $\varphi^+ := \bigwedge_{i+1\le j\le i_R-1}\Gamma.j$
    $\varphi_C := (\bigwedge_{1\le j\le i_L}\Gamma.j) \wedge (\bigwedge_{i_R\le j\le n}\Gamma.j)$
    $\psi := \textsc{Itp}.(\varphi^-, \varphi^+ \wedge \varphi_C).(\mathcal{P})$
    $\Pi.pc_i := \Pi.pc_i \cup \mathsf{Atoms}.(\mathsf{Clean}.\psi)$
return $\Pi$.
```

$$\begin{array}{rl}
(\mathtt{assume}\ (x \neq y) : pc_1); & \langle x,0\rangle \neq \langle y,0\rangle \\
(*x := 0 : pc_2); & \langle *\langle x,0\rangle,3\rangle = 0 \\
(y := x : pc_3); & \langle y,4\rangle = \langle x,0\rangle \wedge \langle *\langle y,4\rangle,5\rangle = \langle *\langle x,0\rangle,3\rangle \\
(\mathtt{assume}(y = x) : pc_4); & \langle y,4\rangle = \langle x,0\rangle \wedge \langle *\langle y,4\rangle,5\rangle = \langle *\langle x,0\rangle,3\rangle \\
(*y := *y+1 : pc_5); & \langle *\langle y,4\rangle,6\rangle = \langle *\langle y,4\rangle,5\rangle + 1 \wedge \\
 & \qquad \mathsf{ite}.(\langle x,0\rangle = \langle y,4\rangle) \\
 & \qquad\quad .(\langle *\langle x,0\rangle,7\rangle = \langle *\langle y,4\rangle,5\rangle + 1) \\
 & \qquad\quad .(\langle *\langle x,0\rangle,7\rangle = (\langle *\langle x,0\rangle,3\rangle)) \\
(\mathtt{assume}(*x = 0) : pc_6) & \langle *\langle x,0\rangle,7\rangle = 0
\end{array}$$

Figure 9. Cutting a PIII trace.

$\varphi^+ = \Gamma.5 = (\langle r,5\rangle = \langle x,4\rangle)$. Furthermore, $\varphi_C^- = (\Gamma.1 \wedge \Gamma.2) = (\langle a,1\rangle = 0 \wedge \langle \phi_x,2\rangle = \langle a,1\rangle)$ and $\varphi_C^+ = (\Gamma.6 \wedge \Gamma.7) = (\langle b,6\rangle = \langle r,5\rangle \wedge (\langle a,1\rangle \neq \langle b,6\rangle - 1)$. The symbols common to φ^- and $\varphi^+ \wedge (\varphi_C^- \wedge \varphi_C^+)$ are $\langle x,4\rangle$ and $\langle \phi_x,2\rangle$, which are the values of x and ϕ_x at that point. The interpolant is $\langle x,4\rangle = \langle \phi_x,2\rangle + 1$, which yields the predicate $x = \phi_x + 1$. Similarly, when we cut the trace at the fifth location, the common variables are $\langle r,5\rangle$ and $\langle \phi_x,2\rangle$, and the cleaned interpolant which summarizes the function's behavior for all calling contexts is $r = \phi_x + 1$. Notice that at this point, which is the last location of the function body, the interpolant is guaranteed to only use the "input" and "output" variables of the function. When cutting the trace at the sixth location, we get the predicate $b = a - 1$, which is again well-scoped for pc_6, where the variables x, ϕ_x, and r are not in scope. It is easy to check that these predicates make the trace abstractly infeasible. □

6 Programs with Pointers

We now consider programs that deal with pointers. As before, we first consider "flat" programs, and then move on to programs with procedures. The only difference with the previous section is in the constraint generation; the algorithm Extract is exactly the same as before, only it uses a different function Con. In this somewhat technical section we show how the constraint generation must be extended to handle pointers in a sound and complete way.

Stores. The classical way to model the store is to use memory expressions and the theory of arrays [13, 26, 27], which comes equipped with two special functions, `sel` and `upd`. The function `sel` takes a memory M and an address a and returns the contents of the address a; the function `upd` takes a memory M, an address a, and a value v, and returns a new memory that agrees with M except that the address a now has value v. The relationship between `sel` and `upd` is succinctly stated by McCarthy's axiom [21]: $\mathtt{sel}(\mathtt{upd}(M,a,v),b) = \mathsf{ite}.(a = b).v.\mathtt{sel}(M,b)$. For a memory variable M and a variable x, define $M.x = \mathtt{sel}(M,x)$, and for an lvalue $*l$, define $M.(*l) = \mathtt{sel}(M,M.l)$. With some slight abuse of notation, we use M in this way to denote a map from lvalues to memory expressions over M. We naturally extend the map M to expressions and formulas of *FOL*. Expressions and formulas appearing in the program do not contain memory variables, `sel`, or `upd`.

Symbolic variables. We must generalize the notion of symbolic variables, which freeze the formal parameters of a function, because now a pointer may be passed to a function and the function may change the value of some cell that is reachable using the pointer. Hence, we have to relate the values of cells at the return of a function with the values the cells had upon the corresponding function call. For a set X of variables, let $\mathsf{Reach}.X = \{*^k x \mid x \in X \text{ and } k \ge 0\}$ be the set of cells that are reachable from X by dereferences. As we do not have recursive types, this set is finite and syntactically computable (k is bounded by the type of x). The set of *symbolic variables* of f is now $sym_f = \{\phi_l \mid l \in \mathsf{Reach}.X_f\}$, where X_f is the set of formal parameters of f. As before, (1) the symbolic variables are local variables of the function which are never written to, and (2) the program is changed, by a syntactic preprocessing pass, so that each lvalue that is reachable from the formal parameters reads its value from the corresponding symbolic variable (*i.e.,* x from ϕ_x, $*x$ from ϕ_{*x} and so on) in the very first statement of the function (using $\mathtt{assume}(\bigwedge_{l\in\mathsf{Reach}.X_f} l = \phi_l)$). As before, for modularity we analyze the function using the symbolic variables, and replace their values with the actual parameters at the call sites [1, 29]. For a symbolic variable ϕ_l, define $(\phi_l \downarrow) = l$.

Constraints for modeling allocation. Suppose there are two variables x and y, each of type `ref int`. When the program begins, and the pointers are allocated, the standard semantics is that their values are not equal. For completeness, this must be explicitly modeled by constraints. We modify the transformation, described earlier, which inserts into every function as first statement an `assume` that copies the symbolic variables into the formal parameters, to include another clause that states that every two distinct lvalues $l_1, l_2 \in \mathsf{Reach}.(V_f \setminus (X_f \cup sym_f))$ of reference type are not equal. Again, as the types are nonrecursive, this clause is quadratic in the size of the function. An example is the first `assume` in the trace of Figure 9.

Constraints for modeling the store with lvalue maps. Using `sel` and `upd` it is straightforward to generate the strongest postconditions for programs with pointers; see Figure 10. Unfortunately, the theory of arrays does not have the interpolant property, thus we cannot get interpolants from TFs that use this theory. For example, the conjunction of $M' = \mathtt{upd}(M,x,y)$ and $(a \neq b) \wedge (\mathtt{sel}(M,a) \neq \mathtt{sel}(M',a)) \wedge (\mathtt{sel}(M,b) \neq \mathtt{sel}(M',b))$ is unsatisfiable, but there is no quantifier-free interpolant in the common set of variables, namely $\{M, M'\}$. We surmount this hurdle by modeling the memory axioms using (generalized) lvalue maps, and by instantiating the array axioms on demand. Recall the definitions of lvalue maps and Upd from Section 5. The set *ChLval* consists of elements *cl* generated by the grammar $cl ::= \langle x,i\rangle \mid \langle cl,i\rangle$, where $i \in \mathbb{N}$. The function Clean of the previous section is extended by $\mathsf{Clean}.\langle x,i\rangle = x$ and $\mathsf{Clean}.\langle cl,i\rangle = *(\mathsf{Clean}.cl)$. Each $cl \in ChLval$ is a special constant that denotes the value of $\mathsf{Clean}.cl$ at some point in the trace. The function Sub of the previous section is extended to all lvalues by $\mathsf{Sub}.\theta.(*^k x) = \langle x, \theta.x\rangle$ if $k = 0$, and $\mathsf{Sub}.\theta.(*^k x) = \langle \mathsf{Sub}.\theta.*^{k-1}x, \theta.(*^k x)\rangle$ otherwise, and extended naturally to expressions, atomic predicates, and formulas.

Constraints for assume operations. Modeling the memory with `sel` and `upd` gives us some relations for free, *e.g.,* from $x = y$

(modeled as $\mathtt{sel}(M,x) = \mathtt{sel}(M,y)$) the equality $*x = *y$ (modeled as $\mathtt{sel}(M,\mathtt{sel}(M,x)) = \mathtt{sel}(M,\mathtt{sel}(M,y))$) follows by congruence. We explicitly state these implied equalities when generating constraints, by closing a predicate with the operator $\mathsf{clos}^*.\mathtt{true}$: $FOL \rightarrow FOL$, where

$$\mathsf{clos}^*.b.p = \begin{cases} (\mathsf{clos}^*.b.p_1)\ \mathsf{op}\ (\mathsf{clos}^*.b.p_2) & \text{if } p \equiv (p_1\ \mathsf{op}\ p_2), \\ \neg(\mathsf{clos}^*.(\neg b).p_1) & \text{if } p \equiv (\neg p_1), \\ p \wedge \bigwedge_{0 \le k \le N}((*^k l_1) = (*^k l_2)) & \text{if } p \equiv (l_1 = l_2) \text{ and} \\ & \quad b = \mathtt{true}, \\ p & \text{otherwise,} \end{cases}$$

provided $\mathsf{typ}.l_1 = \mathsf{typ}.l_2 = \mathtt{ref}^N\mathtt{int}$. The formula $\mathsf{clos}^*.\mathtt{true}.p$ explicates all equalities inferred by the memory axioms from the formula p. When generating the constraints for $\mathtt{assume}(p)$, we first "close" p using clos^*, and then generate constraints for the result. Consider, for example, the constraint for the fourth command in Figure 9. For any formula p that can appear in an $\mathtt{assume}$, we have $M.p \Leftrightarrow M.(\mathsf{clos}^*.\mathtt{true}.p)$ in the theory of arrays. Using this equivalence, we can show the following lemma, which tells us that the constraints have been modeled adequately. For a program $\mathbb{P}$, an lvalue $*^k x$ is well-typed in $\mathbb{P}$ if $\mathsf{typ}.x = \mathtt{ref}\ ^N\mathtt{int}$ for some $N \ge k$, *i.e.,* if x has type $\mathtt{ref\ int}$, then $*x$ is well-typed but not $***x$. A formula p is well-typed w.r.t. $\mathbb{P}$ if (1) it does nor contain memory variables, $\mathtt{sel}$, or $\mathtt{upd}$, and (2) each lvalue that occurs in p is well-typed in P.

LEMMA 1. *For a program $\mathbb{P}$, two formulas $p, p' \in FOL$ that are well-typed w.r.t. $\mathbb{P}$, and an lvalue map θ, the condition $M.p$ implies $M.p'$ iff $\mathsf{Sub}.\theta.(\mathsf{clos}^*.\mathtt{true}.p)$ implies $\mathsf{Sub}.\theta.p'$.*

Constraints for assignments. When assigning to $*l_1$ we must explicate that for all lvalues $*l_2$ such that $l_1 = l_2$, the value of $*l_2$ is updated as well. Let Equate be a function that takes a pair of lvalue maps (θ_1, θ_2) and a pair of expressions (l_1, l_2), and generates equalities between the names of l_1 and its transitive dereferences under θ_1, and the names of l_2 and its transitive dereferences under θ_2. Formally,

$$\mathsf{Equate}.(\theta_1,\theta_2).(l_1,l_2) = \bigwedge_{0 \le k \le N} (\mathsf{Sub}.\theta_1.(*^k l_1) = \mathsf{Sub}.\theta_2.(*^k l_2)),$$

where $\mathsf{typ}.l_1 = \mathsf{typ}.l_2 = \mathtt{ref}^N\mathtt{int}$. Define the function EqAddr, which takes a pair of lvalues and returns a formula that is true when the lvalues have the same address, by $\mathsf{EqAddr}.(*^{k_1}x_1, *^{k_2}x_2) = \mathtt{false}$ if $k_1 = 0$ or $k_2 = 0$, and $\mathsf{EqAddr}.(*^{k_1}x_1, *^{k_2}x_2) = (*^{k_1-1}x_1 = *^{k_2-1}x_2)$ otherwise. For a function f (which is $\mathtt{main}$ for flat programs), let $Lvals.f = \mathsf{Reach}.V_f$. For an lvalue l and a function f, let $\mathsf{Alias}.f.l = \{l' \mid l' \in Lvals.f$ and l' may be aliased to $l\}$. We only require that $\mathsf{Alias}.f.l$ overapproximates the set of lvalues that can actually alias l.

Finally, we define the function Asgn, which generates appropriate constraints for an assignment $l := e$. The function Asgn takes a function name f, an lvalue map θ, and a pair (l,e), where l is an lvalue and e the expression that is being written into l, and returns a pair (θ', φ') of an updated lvalue map θ' and a formula φ'. Define $\theta' = \mathsf{Upd}.\theta.S$, where $S = \{*^k l' \mid l' \in (\mathsf{Alias}.l) \cup \{l\}$ and $k \ge 0\}$, and define

$$\varphi' = \mathsf{Equate}.(\theta',\theta).(l,e) \wedge \bigwedge_{l' \in \mathsf{Alias}.f.l} \begin{pmatrix} \mathsf{ite}. & (\mathsf{Sub}.\theta.(\mathsf{EqAddr}.(l,l'))). \\ & (\mathsf{Equate}.(\theta',\theta).(l',e)). \\ & (\mathsf{Equate}.(\theta',\theta).(l',l')) \end{pmatrix}.$$

The first conjunct of φ' states that l gets a new value e, and all transitive dereferences of l and e are "equated" (*i.e.,* $*l$ gets the new value $*e$, and so on). The big second conjunct of φ' states how the potential aliases l' of l are updated: if l and l' have the same address, then the new value of l' (given by $\mathsf{Sub}.\theta'.l'$) is equated with e; otherwise the new value of l' is equated with the old value of l' (given by $\mathsf{Sub}.\theta.l'$). This generalizes Morris' definition for the strongest postcondition in the presence of pointers [24].

LEMMA 2. *Let $l := e$ be an assignment in a program $\mathbb{P}$, let $\varphi = \mathsf{SP}.\mathtt{true}.(l := e)$, and let $(\theta', \varphi') = \mathsf{Con}.(\theta_0, \mathtt{true}).t$ for some lvalue map θ_0. For every formula $p \in FOL$ that is well typed w.r.t. $\mathbb{P}$, the formula φ implies $M.p$ in the theory of arrays iff φ' implies $\mathsf{Sub}.\theta'.p$.*

6.1 Flat Programs: PIII

The first three rows of Figure 10 give the definition of the operator SP using the theory of arrays, as well as the generated constraints. The definition of $\mathsf{SP}_\sqcap$ is the same as before, except that the new SP operator is used. Notice that the "current" memory is always represented by M. For assignments, SP states that the current memory M is now an updated version of the old memory, which is renamed M'. We use Asgn to generate the appropriate constraints for dealing with the possible alias scenarios. For assume operations, SP is defined as before, except that the constraint generated is on the "closure" of the predicate using clos^*. Constraints for sequences are obtained by composition. The size of the constraints is quadratic in the size of the trace. By induction over the length of the trace, splitting cases on the kind of the last operation, and using Lemmas 1 and 2, we can prove the following theorem.

THEOREM 2. *Given a trace t of a program $\mathbb{P}$, let $(\theta', \Gamma) = \mathsf{Con}.(\theta_0, \Gamma_0).t$, let $\varphi_r = \mathsf{SP}.\mathtt{true}.t$, and let $\varphi = \bigwedge_{1 \le i \le n} \Gamma.i$. For every formula $p \in FOL$ that is well-typed w.r.t. $\mathbb{P}$, the formula φ_r implies $M.p$ in the theory of arrays iff φ implies $\mathsf{Sub}.\theta'.p$. Hence, the trace t is feasible iff the formula φ is satisfiable.*

Given the new definition of Con, the algorithm for predicate discovery is the same as $\mathsf{Extract}$ (Algorithm 1), and Theorem 1 holds.

EXAMPLE 4: The right column in Figure 9 shows the constraints for the trace on the left. For readability, we omit unsatisfiable and uninteresting disjuncts (for the second and third commands). At the fourth cut-point of this trace, the common variables are $\langle *\langle y,4\rangle, 3\rangle$, $\langle y,4\rangle$, $\langle x,0\rangle$, and $\langle *\langle x,0\rangle, 3\rangle$, which denote the values of $*y$, y, x, and $*x$ at that point in the trace. The interpolant for this cut is $\langle *\langle y,2\rangle, 3\rangle = 0$, which gives the predicate $*y = 0$ for the location pc_4. □

6.2 General Programs: PIV

The main difficulty when dealing with functions and pointers is in handling the semantics of calls and returns, because the callee may be passed pointers into the local space of the caller. The complexity arises when we wish to abstract functions polymorphically [1], because then we have to summarize all effects that the callee may have had on the caller's store at the point of return. One way to do this is to imagine the callee starting with a *copy* of the caller's store and, upon return, the caller refreshing his store appropriately using the callee's store. As we shall see, the difficulty is only in modeling this appropriately with strongest postconditions. Following that, it

t	$\mathsf{SP}.\varphi.t$	$\mathsf{Con}.(\theta,\Gamma).t$
$(l := e : pc_i)$	$\exists M'.(\varphi[M'/M] \wedge M = \mathtt{upd}(M', M'.l, M'.e))$ where M' is a fresh store	$(\theta', \Gamma[i \mapsto \varphi])$ where $(\theta', \varphi) = \mathsf{Asgn}.f.\theta.(l, e)$ f is the function in which pc_i belongs
$(\mathtt{assume}(p) : pc_i)$	$\varphi \wedge M.p$	$(\theta, \Gamma[i \mapsto \mathsf{Sub}.\theta.(\mathsf{clos}^*.\mathtt{true}.p)])$
$t_1;t_2$	$\mathsf{SP}.(\mathsf{SP}.\varphi.t_1).t_2$	$\mathsf{Con}.(\mathsf{Con}.(\theta,\Gamma).t_1).t_2$
$(f(\vec{y}) : pc_i);$ $t_1;$ $(\mathtt{return} : pc_j)$	$\exists M', M_I, M_O.sym_f.$ $\varphi[M'/M]$ $\wedge M' = M_I \wedge \varphi_{ld}$ $\wedge\ \varphi'[M_O/M]$ $\wedge\ \mathsf{New}.(L,P,R').(M_O, M_I)$ $\wedge\ M = \mathsf{copy}(M', M_O, R)$ where M', M_I, M_0 are fresh stores $\varphi_{ld} = \bigwedge_{\phi \in sym_f} \phi = M_I.(\phi\downarrow)[y/x]$ $\vec{x}$ are the formals of f $R = \{*\phi \mid \phi \in sym_f\}$ $\varphi' = \exists V_f^-.\mathsf{SP}.\mathtt{true}.t_1$ $V_f^- = V_f \setminus sym_f,$ L = lvalues in scope at pc_2 $P = sym_f$ and $R' = \mathsf{Reach}.P$	(θ', Γ') where $\theta^I = \mathsf{Upd}.\theta.(\mathsf{Reach}.V_f)$ $\varphi_{ld} = \bigwedge_{\phi \in sym_f} \mathsf{Equate}.(\theta^I, \theta).(\phi, (\phi\downarrow)[y/x])$ $\Gamma_I = \Gamma[i \mapsto \varphi_{ld}]$ $(\theta^O, \Gamma_O) = \mathsf{Con}.(\theta^I, \Gamma_I).t_1$ L, P, R, R' as for SP $\Gamma' = \Gamma_O[j \mapsto \mathsf{New}.(L,P,R').(\theta, \theta^O) \wedge \mathsf{copy}.(\theta, \theta', \theta^O).(L, R)]$ $\theta' = \mathsf{Upd}.\theta.L$

Figure 10. Postconditions and constraints for PIII and PIV traces.

$(*y := 0 : pc_1);$	$\langle *\langle y,0\rangle, 1\rangle = 0$
$(inc(y) : pc_2);$	$\langle \phi_x, 1\rangle = \langle y, 0\rangle \wedge \langle \phi_{*x}, 2\rangle = \langle *\langle y,0\rangle, 1\rangle$
$(\mathtt{assume}\ (x = \phi_x$ $\wedge *x = \phi_{*x}) : pc_3);$	$\langle x, 4\rangle = \langle \phi_x, 1\rangle$ $\wedge \langle *\langle x,4\rangle, 5\rangle = \langle *\langle \phi_x, 1\rangle, 7\rangle$ $\wedge \langle *\langle x,4\rangle, 5\rangle = \langle \phi_{*x}, 2\rangle$
$(*x := *x + 1 : pc_4);$	$\langle *\langle x,4\rangle, 6\rangle = \langle *\langle x,4\rangle, 5\rangle + 1$ $\mathsf{ite}.(\langle x,4\rangle = \langle \phi_x, 1\rangle)$ $.(\langle *\langle \phi_x,1\rangle, 8\rangle = \langle *\langle x,4\rangle, 5\rangle + 1)$ $.(\langle *\langle \phi_x,1\rangle, 8\rangle = \langle *\langle \phi_x,1\rangle, 7\rangle$
$(\mathtt{return}\ : pc_5);$	$\mathsf{ite}.(\langle y,0\rangle = \langle \phi_x, 1\rangle)$ $.(\langle *\langle y,0\rangle, 9\rangle = \langle *\langle \phi_x,1\rangle, 8\rangle)$ $.(\langle *\langle y,0\rangle, 9\rangle = \langle *\langle y,0\rangle, 1\rangle).$
$(\mathtt{assume}(*y \neq 1) : pc_6)$	$\langle *\langle y,0\rangle, 9\rangle \neq 1$

Figure 11. Cutting a PIV trace.

is straightforward to generate the constraints, and the method for learning predicates is again Algorithm 2, only now using the new definition of Con.

As we allow functions to pass pointers into the local store as arguments, to keep the exposition simple, we can assume w.l.o.g. that (1) there are no global variables (these can be modeled by passing references), and (2) there is no explicit "return" variable; instead, return values are passed by updating the contents of some cell that is passed in as a parameter.

EXAMPLE 5: Consider the trace on the left in Figure 11. The caller (`main`) passes a pointer y to the store to the callee `inc`. The callee updates the memory address $*y$ (which is called $*x$ in `inc`). There are two symbolic variables ϕ_x and ϕ_{*x} in `inc`, corresponding to the formal parameter x and its dereference. The `assume` at location pc_3 loads the symbolic variables into the formal parameters. □

Soundness. Every cell in the caller's store which is modified by the callee must be reachable from a parameter passed to the callee, *e.g.,* the cell pointed to by y in the caller is the same as the cell pointed to by x when the function `inc` is called, hence upon return the value of $*y$ should be the value of $*(\phi_x)$ (as in the interim, the callee may have changed x). Every cell of the caller which is unreachable from the parameters passed to the callee remains unchanged as a result of the call. This is modeled in the SP semantics by copying the contents of the callee's store, at the locations reachable from the passed parameters, into the caller's store. The locations that are reachable from the passed parameters are frozen in the symbolic variables of the callee. It can be shown that this is sound for our language. To express the copying in SP, we use the operator copy: for a destination memory M_d, a source memory M_s, and a dereferenced lvalue $*l$, the expression $\mathsf{copy}(M_d, M_s, *l)$ is the result of updating M_d such that all cells reachable from l have the same value as in M_s. Formally, we define $\mathsf{copy}(M_d, M_s, *l)$ as

$$\begin{cases} \mathtt{upd}(M_d, M_d.l, M_s.(*l)) & \text{if } \mathsf{typ}.*l = \mathtt{int}, \\ \mathsf{copy}(\mathtt{upd}(M_d, M_d.l, M_s.(*l)), M_s, **l) & \text{otherwise.} \end{cases}$$

We never need to copy into a variable x. The function copy is extended to a sequence of lvalues $\vec{l} = (*l :: \vec{l'})$ by $\mathsf{copy}(M_d, M_s, \vec{l}) = \mathsf{copy}(\mathsf{copy}(M_d, M_s, *l), M_s, \vec{l'})$. It can be shown that the result is independent of the order of lvalues. Hence, we can consider the operator copy to be defined over *sets* of lvalues. We can mimic "copying" by lvalue maps as follows. Given three lvalue maps θ_d, θ'_d, and θ_s, an lvalue $*l$, and a sequence R of lvalues, define $\mathsf{copy}.(\theta_d, \theta'_d, \theta_s).(*l, R) =$

$$\begin{cases} \mathsf{Sub}.\theta'_d.*l = \mathsf{Sub}.\theta_d.*l & \text{if } R \equiv \cdot, \\ \mathsf{ite}.(\mathsf{Sub}.\theta'_d.l = \mathsf{Sub}.\theta_d.\phi). & \\ \quad (\mathsf{Equate}.(\theta'_d, \theta_s).(*l, *\phi)). & \\ \quad \mathsf{copy}.(\theta_d, \theta'_d, \theta_s).(*l, R') & \text{if } R \equiv \phi :: R'. \end{cases}$$

Finally, for a set L of lvalues, define $\mathsf{copy}.(\theta_d, \theta'_d, \theta_s).(L, R) = \bigwedge_{*l \in L} \mathsf{copy}.(\theta_d, \theta'_d, \theta_s).(*l, R)$.

When a function returns, we update all local variables of the caller. We set θ_d to the lvalue map of the caller *before* the call, and $\theta'_d = \mathsf{Upd}.\theta_d.L$ is the lvalue map of the caller *after* the call, where $L = \mathsf{Reach}.\{*x \mid x \text{ is a local variable of the caller}\}$ is the set of lvalues of the caller that can change (no local variable x can change as the result of a call; only dereferences can change). We set θ_s to the lvalue map of the callee upon return, and $R = sym_f$ is the set of cells that were passed to the callee, and hence must be copied back into the caller. It can be checked that the formulas resulting from different permutations of R are equivalent.

Completeness. For completeness, we must ensure that we properly model the semantics of allocation. It can be shown that it suffices to ensure that every cell that is being "returned" by the callee (*i.e.,* reachable from a symbolic variable of the callee) is either a cell passed to it (*i.e.,* equal to some symbolic variable) or is brand new (*i.e.,* different from $\mathsf{Reach}.V_f$, the set of cells known to the caller). If a cell is different from those of the caller, then transitivity of equality and the same check on all subsequent returns ensures that the cell is different from all previously allocated ones. The check is encoded with the operator New. For an lvalue l of reference type, a set L of lvalues, and two stores M_l and M_L, define $\mathsf{diff}.(l,L).(M_l,M_L) = \wedge_{l' \in L} \neg(M_l.l = M_L.l')$. Given three sets L, P, and R of lvalues, and a pair M_O and M_I of stores, define $\mathsf{New}.(L,P,R).(M_O,M_I) = (\wedge_{r \in R}(\mathsf{diff}.(r,P).(M_O,M_I) \Rightarrow \mathsf{diff}.(r,L).(M_O,M_I))$. Here $L = \mathsf{Reach}.V_{f'}$ is the set of local variables of the caller f', and $P = sym_f$ is the set of cells passed to the callee f, and $R = \mathsf{Reach}.sym_f$ is the set of cells returned by the callee. The store M_O is the store upon return from f, and M_I was the store upon entry to f. The formula says that for every cell r that is returned, if r is different from all cells passed to f, then r is different from all local cells L of the caller f'. This is generalized to lvalue maps as follows: $\mathsf{diff}.(l,L).(\theta_l,\theta_L) = \wedge_{l' \in L} \neg(\mathsf{Sub}.\theta_l.l = \mathsf{Sub}.\theta_L.l')$ and $\mathsf{New}.(L,P,R).(\theta,\theta') = (\wedge_{r \in R}(\mathsf{diff}.(r,P).(\theta,\theta')) \Rightarrow \mathsf{diff}(r,L).(\theta,\theta'))$.

Strongest postconditions and constraints. Using these functions, we can generate the strongest postconditions and the constraints as shown in Figure 10. Assignments, assume operations, and sequencing is handled as before; we describe here only function calls. For SP, we rename the caller's store to M' as it will change as a result of the call. We pass a memory M_I equal to M' to the callee, equate the actual parameters with the symbolic variables, and compute SP of the callee. Then we rename the memory returned by the callee to M_O, and copy back the local store modified by the call into M' to get the current memory M. Additionally, we add distinctness constraints to model allocation. The definition of $\mathsf{SP}_{\sqcap}$ is similar to the one before: before it was the predicate abstraction of SP, using $\mathsf{SP}_{\sqcap}$ to analyze the call body; now it is the predicate abstraction (using the predicates at the return location) of the new SP, using $\mathsf{SP}_{\sqcap}$ recursively for the call body. The correctness of the constraint generation is stated in Theorem 2. The size of the constraints is cubic in the size of the trace. Given the new definition of Con, the method for predicate discovery is Algorithm 2, and Theorem 1 holds.

EXAMPLE 6: Consider the trace in Figure 11. The right column shows the constraints that correspond to each command. The constraint from the assignment $*y = 0$ is $\langle *\langle y,1\rangle,1\rangle = 0$. First, the constraint for the call command is the clause φ_{ld}, which loads the actual parameters into to the symbolic constants for `inc`. The first command in the body loads the values from the symbolic constants into the formal parameters; notice that we take the "closure". We then build the constraints for the increment operation. Now $L = \{y, *y\}$, $P = \{\phi_x, \phi_{*x}\}$, $R = \{*\phi_x\}$, and $R' = \{\phi_x, *\phi_x, \phi_{*x}\}$. The constraint $\mathsf{New}.(L,P,R).(\theta,\theta^O)$ simplifies to `true`, because $*\phi_x$ is not a reference type, and $\langle \phi_x, 1\rangle = \langle y, 0\rangle$, *i.e.*, it is a cell that was passed to f. Let θ' be the map updating θ so that $*y$ is mapped to 9. Finally, the copy-back constraint $\mathsf{copy}.(\theta,\theta',\theta^O).(L,R)$ is shown to the right of the return. At the end, the assume operation generates the constraint $\langle *\langle y,1\rangle,3\rangle \neq 1$. The set of generated constraints is unsatisfiable. Consider the fourth cut-point of this trace, *i.e.*, up to and including the increment operation. The common variables are $\langle \phi_x, 1\rangle$, $\langle *\langle \phi_x,1\rangle,8\rangle$, and $\langle \phi_{*x}, 2\rangle$; they denote the current values of ϕ_x, $*\phi_x$, and ϕ_{*x}, respectively. The interpolant for this cut is $\langle *\langle \phi_x,1\rangle,8\rangle = \langle \phi_{*x},2\rangle + 1$, which upon cleaning gives the predicate $*\phi_x = \phi_{*x} + 1$. This predicate asserts that the present cell pointed to by ϕ_x has a value 1 greater than the cell $*x$ had upon entry to f. □

7 Experiments

We have implemented the interpolation based abstraction refinement scheme in the software model checker BLAST [19]. The algorithm for generating interpolants uses the VAMPYRE proof-generating theorem prover.[2] For efficiency, we have implemented several optimizations of the basic procedure described in this paper. First, we treat sequences of assignments atomically. Second, we do not cut at every point of a spurious error trace. Instead, we perform a preliminary analysis which identifies a subset of the constraints that imply the infeasibility of the trace, and only consider the instructions that generate these constraints as cut-points. It is easy to check that the optimized procedure is still complete. For pointers, we only generate constraints between expressions of the same type. With these optimizations, we find, for example, that the two predicates $a \leq c - 2$ and $b > 0$ suffice to prove the trace in Example 1 infeasible. These are fewer predicates than those generated by the heuristically optimized predicate discovery scheme of [3].

We ran interpolation based BLAST on several Windows NT device drivers, checking a property related to the handling of I/O Request packets. The property is a finite-state automaton with 22 states [2]. The results, obtained on an IBM ThinkPad T30 laptop with a 2.4 GHz Pentium processor and 512MB RAM, are summarized in Table 1. We present three sets of numbers: 'Previous' gives the times for running the previous version of BLAST, without interpolants; 'Craig' uses interpolants to discover predicates, and drops predicates that are out of scope, but it does not track different sets of predicates at individual program locations; 'Craig+Locality' uses interpolants and tracks only the relevant predicates at each program location. The previous version of BLAST timed out after several hours on the drivers `parport` and `parclass`. We found several violations of the specification in `parclass`. The numbers in the table refer to a version of `parclass` where the cases that contain errors are commented out. Both 'Craig' and 'Craig+Locality' perform better than the previous version of BLAST. When started with the empty set of initial predicates, 'Craig' is faster than 'Craig+Locality', because 'Craig+Locality' may rediscover the same predicate at several different program locations. However, since the predicates are tracked extremely precisely (the average number of predicates at a program location is much smaller than the total number of predicates required), 'Craig+Locality' uses considerably less memory, and subsequent runs (for example, for verifying a modified version of the program [18], or for generating PCC-style proofs [17]) are faster, and the proof trees smaller.

8 References

[1] T. Ball, T. Millstein, and S.K. Rajamani. Polymorphic predicate abstraction. *ACM Transactions on Programming Languages and Systems*, 2003.

[2] T. Ball and S.K. Rajamani. Personal communication.

[2] VAMPYRE is available from http://www.eecs.berkeley.edu/∼rupak/Vampyre.

Program	LOC	Previous		Craig			Craig+Locality			
		Disc	Reach	Disc	Reach	Preds	Disc	Reach	Preds	Avg/Max
`kbfiltr`	12301	1m12s	0m30s	0m52s	0m22s	49	3m48s	0m10s	72	6.5/16
`floppy`	17707	7m10s	3m59s	7m56s	3m21s	156	25m20s	0m46s	240	7.7/37
`diskperf`	14286	5m36s	3m3s	3m13s	1m18s	86	13m32s	0m27s	140	10/31
`cdaudio`	18209	20m18s	4m55s	17m47s	4m12s	196	23m51s	0m52s	256	7.8/27
`parport`	61777	-	-	-	-	-	74m58s	2m23s	753	8.1/32
`parclass`	138373	-	-	42m24s	9m1s	251	77m40s	1m6s	382	7.2/28

Table 1. Experimental results using BLAST: 'm' stands for minutes, 's' for seconds; 'LOC' is the number of lines of preprocessed code; 'Disc' is the total running time of the verification starting with the empty set of predicates; 'Reach' is the time to perform the reachability analysis only, given all necessary predicates; 'Preds' is the total number of predicates required, and Avg (Max) is the average (maximum) number of predicates tracked at a program location; the symbol '-' indicates that the tool does not finish in 6 hours.

[3] T. Ball and S.K. Rajamani. Generating abstract explanations of spurious counterexamples in C programs. Technical Report MSR-TR-2002-09, Microsoft Research, 2002.

[4] T. Ball and S.K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL 02: Principles of Programming Languages*, pages 1–3. ACM, 2002.

[5] R. Bodik, R. Gupta, and M.L. Soffa. Refining dataflow information using infeasible paths. In *FSE 97: Foundations of Software Engineering*, LNCS 1301, pages 361–377. Springer, 1997.

[6] W.R. Bush, J.D. Pincus, and D.J. Sielaff. A static analyzer for finding dynamic programming errors. *Software Practice and Experience*, 30:775–802, 2000.

[7] S. Chaki, E.M. Clarke, A. Groce, and O. Strichman. Predicate abstraction with minimum predicates. In *CHARME 03: Correct Hardware Design and Verification*, LNCS 2860, pages 19–34. Springer, 2003.

[8] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV 00: Computer-Aided Verification*, LNCS 1855, pages 154–169. Springer, 2000.

[9] W. Craig. Linear reasoning. *J. Symbolic Logic*, 22:250–268, 1957.

[10] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and F.K. Zadek. Efficiently computing static single assignment form and the program dependence graph. *ACM Transactions on Programming Languages and Systems*, 13:451–490, 1991.

[11] M. Das, S. Lerner, and M. Seigle. ESP: path-sensitive program verification in polynomial time. In *PLDI 02: Programming Language Design and Implementation*, pages 57–68. ACM, 2002.

[12] C. Flanagan. Automatic software model checking using CLP. In *ESOP 03: European Symposium on Programming*, LNCS 2618, pages 189–203. Springer, 2003.

[13] C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI 02: Programming Language Design and Implementation*, pages 234–245. ACM, 2002.

[14] C. Flanagan and J.B. Saxe. Avoiding exponential explosion: generating compact verification conditions. In *POPL 01: Principles of Programming Languages*, pages 193–205. ACM, 2001.

[15] J.S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *PLDI 02: Programming Language Design and Implementation*, pages 1–12. ACM, 2002.

[16] D. Gries. *The Science of Programming*. Springer, 1981.

[17] T.A. Henzinger, R. Jhala, R. Majumdar, G.C. Necula, G. Sutre, and W. Weimer. Temporal-safety proofs for systems code. In *CAV 02: Computer-Aided Verification*, LNCS 2404, pages 526–538. Springer, 2002.

[18] T.A. Henzinger, R. Jhala, R. Majumdar, and M.A.A. Sanvido. Extreme model checking. In *International Symposium on Verification*, LNCS. Springer, 2003.

[19] T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL 02: Principles of Programming Languages*, pages 58–70. ACM, 2002.

[20] J. Krajicek. Interpolation theorems, lower bounds for proof systems, and independence results for bounded arithmetic. *J. Symbolic Logic*, 62:457–486, 1997.

[21] J. McCarthy and J. Painter. Correctness of a compiler for arithmetic expressions. In *Proc. Symposia in Applied Mathematics*. American Mathematical Society, 1967.

[22] K.L. McMillan. Interpolation and SAT-based model checking. In *CAV 03: Computer-Aided Verification*, LNCS 2725, pages 1–13. Springer, 2003.

[23] J.C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.

[24] J. M. Morris. A general axiom of assignment. In *Theoretical Foundations of Programming Methodology*, Lecture Notes of an International Summer School, pages 25–34. D. Reidel Publishing Company, 1982.

[25] M. Musuvathi, D.Y.W. Park, A. Chou, D.R. Engler, and D.L. Dill. CMC: A pragmatic approach to model checking real code. In *OSDI 02: Operating Systems Design and Implementation*. ACM, 2002.

[26] G.C. Necula. Proof carrying code. In *POPL 97: Principles of Programming Languages*, pages 106–119. ACM, 1997.

[27] G. Nelson. Techniques for program verification. Technical Report CSL81-10, Xerox Palo Alto Research Center, 1981.

[28] P. Pudlak. Lower bounds for resolution and cutting plane proofs and monotone computations. *J. Symbolic Logic*, 62:981–998, 1997.

[29] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL 95: Principles of Programming Languages*, pages 49–61. ACM, 1995.