Sound, Complete and Scalable Path-Sensitive Analysis *

Isil Dillig Thomas Dillig Alex Aiken
Computer Science Department
Stanford University
{isil, tdillig, aiken}@cs.stanford.edu

Abstract

We present a new, precise technique for fully path- and context-sensitive program analysis. Our technique exploits two observations: First, using quantified, recursive formulas, path- and context-sensitive conditions for many program properties can be expressed exactly. To compute a closed form solution to such recursive constraints, we differentiate between *observable* and *unobservable* variables, the latter of which are existentially quantified in our approach. Using the insight that unobservable variables can be eliminated outside a certain scope, our technique computes satisfiability- and validity-preserving closed-form solutions to the original recursive constraints. We prove the solution is as precise as the original system for answering may and must queries as well as being small in practice, allowing our technique to scale to the entire Linux kernel, a program with over 6 million lines of code.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification

General Terms Languages, Reliability, Verification, Experimentation

Keywords Static analysis, path- and context-sensitive analysis, strongest necessary/weakest sufficient conditions

1. Introduction

Path-sensitivity is an important element of many program analysis applications, but existing approaches exhibit one or both of two difficulties. First, so far as we know, there are no prior scalable techniques that are also sound and complete for a language with recursion. Second, even in implementations of incomplete methods, interprocedural path-sensitive conditions can become unwieldy and expensive to compute. Existing approaches deal with these problems by some combination of heuristics, accepting limited scalability, and possible non-termination of the analysis (see Section 2 for discussion of related work).

In this paper, we give a new approach that addresses both issues. Our method is sound and complete; for the class of program properties we address, we can decide all path- and context-sensitive queries. We also demonstrate the performance and scalability of our approach with experiments on the entire Linux kernel, a program with over 6MLOC. Two ideas underpin our technique:

- We can write constraints capturing the exact path- and context-sensitive condition under which a program property holds. Unfortunately, we do not know how to solve these constraints.
- However, to answer *may* (might a property hold?) or *must* (must a property hold?) queries, it is equivalent to decide the question for a particular necessary or sufficient condition extracted from the exact constraints. While non-trivial, the constraints for these necessary/sufficient conditions can be solved, and furthermore, the necessary/sufficient conditions are typically much smaller than the original constraints, improving scalability. Since these necessary and sufficient conditions are satisfiability and validity preserving respectively, they involve no loss of precision for answering may and must queries about program properties.

In practice, program analysis systems ask may or must queries about program properties. Our approach effectively takes advantage of which kind of query is to be asked (may or must) to specialize a general representation of the exact path- and context-sensitive condition to a form where the query can be decided. The completeness of our approach is only guaranteed if the base domain of abstract values is finite; for infinite domains our method is still sound, but not necessarily complete. Thus, for example, our approach can be used for sound and complete context- and path-sensitive type state, type qualifier, or dataflow properties, but not, for example, arbitrary shape properties.

Our approach is best illustrated using an example. Consider the following function:

```
bool queryUser(bool featureEnabled) {
   if(!featureEnabled) return false;
   char userInput = getUserInput();
   if(userInput == 'y') return true;
   if(userInput=='n') return false;
   printf("Input must be y or n! Please try again");
   return queryUser(featureEnabled);
}
```

Under what condition does `queryUser` return `true`? Informally, the argument `featureEnabled` must be `true`, and the user input on some recursive call must be $'\mathtt{y}'$. Our system formalizes this intuition by computing a constraint $\Pi_{\alpha,\mathtt{true}}$ characterizing the condition under which `queryUser`, given argument α, returns `true`:

$$\Pi_{\alpha,\mathtt{true}} = \exists\beta.(\alpha = \mathtt{true}) \wedge (\beta = {'\mathtt{y}'} \vee (\neg(\beta = {'\mathtt{n}'}) \wedge \Pi_{\alpha,\mathtt{true}}[\mathtt{true}/\alpha])) \quad (*)$$

* This work was supported by grants from DARPA, NSF (CCF-0430378, CNS-050955, CNS-0716695, and SA4899-10808PG-1), and equipment grants from Dell and Intel.

PLDI'08, June 7–13, 2008, Tucson, Arizona, USA.

This constraint is recursive, which is not surprising given that `queryUser` is a recursive function. The formula under the existential quantifier encodes the conditions under which the function body evaluates to `true`:

1. `featureEnabled` must be `true` (clause: $\alpha = \texttt{true}$),
2. the user input is $'\texttt{y}'$ (clause: $\beta = '\texttt{y}'$),
3. or the user input is not $'\texttt{n}'$ (clause: $\neg(\beta = '\texttt{n}')$) and the recursive call returns `true` (clause: $\Pi_{\alpha,\texttt{true}}[\texttt{true}/\alpha]$). Because the function argument must be `true` if the recursive call is reached, the argument to the recursive call is also `true`, which is expressed by the substitution $[\texttt{true}/\alpha]$.

The equation above illustrates the main features of our constraint language. Primitive constraints include comparing variables representing program values with constants ($\alpha = \texttt{true}$), and compound constraints can be built with the usual boolean connectives $\wedge$, $\vee$, and $\neg$. The translation between actual and formal arguments in a function call is represented, as usual, by a substitution ($[\texttt{true}/\alpha]$), but note the substitution is part of the constraint. The most unusual feature is existential quantification. The quantified variable β intuitively captures the unknown user input: we do not know statically the value of β, just that it has some value—i.e., it exists. The quantifier also captures the scope of the user input, namely that each input is used for one recursive call. We refer to the existentially bound variables as *unobservable variables*. The values of unobservable variables cannot be expressed in terms of the inputs to a function; hence unobservable variables are not visible outside of the procedure invocation in which they are used.

One division between program analysis approaches is between those that are *whole-program* (requiring the entire program to perform analysis) and those that are *modular* (can analyze parts of a program in isolation). A standard approach to modular analysis is to use formulas to represent program states, with free variables in the formulas capturing the unknown state of the partial program's environment. Our motivation for distinguishing unobservable variables is that that they commonly arise in modular program analysis systems. For example, the results of unknown functions (functions unavailable for analysis) are unobservable; some system state may be hidden (e.g., the garbage collector's free list, operating system's process queue, etc.) that can be modeled by unobservable variables, or a static analysis may itself introduce unobservables to represent imprecision in the analysis (e.g., selecting an unknown element of an array). Unobservable variables are useful within their natural scope, for example, tests on unobservables can possibly be proven mutually exclusive (e.g., testing whether the result of a `malloc` call is null, and then testing whether it is non-null). A key observation is that outside of that scope unobservables provide no additional information, at least for answering may or must queries, and can be eliminated. By distinguishing unobservable values, we help separate what is essential to path-sensitive analysis from inherent, but orthogonal, sources of imprecision for any analysis. Thus, the distinction between observables and unobservables is what enables us to prove both soundness and completeness for our constraint resolution algorithm.

The combination of existential quantification and recursion is more expressive than it may first appear. In particular, in the previous example, it captures that the user input may be different on different recursive calls. To see this, consider the constraint written with the recursion fully (infinitely) unfolded:

$$\begin{aligned}\Pi_{\alpha,\texttt{true}} = \quad & \exists\beta.(\alpha = \texttt{true}) \wedge (\beta = '\texttt{y}' \vee \neg(\beta = '\texttt{n}') \wedge \\ & \exists\beta'.(\texttt{true} = \texttt{true}) \wedge (\beta' = '\texttt{y}' \vee \neg(\beta' = '\texttt{n}') \wedge \\ & \exists\beta''.(\texttt{true} = \texttt{true}) \wedge (\beta'' = '\texttt{y}' \vee \neg(\beta'' = '\texttt{n}') \wedge \\ & \ldots\end{aligned}$$

For clarity, we have performed the substitution $[\texttt{true}/\alpha]$ on the unfolded constraints. More interestingly, the quantified variables are renamed to emphasize each recursive call has an independent user input. Thus, the original recursive constraint captures the full interprocedural, path-sensitive condition under which the result of a call to `getUserInput` is `true`. The first part of our work, which includes the predicate language, inference algorithm for constraints, and the class of program properties for which we can compute sound and complete path- and context-sensitive conditions, is discussed in Section 3.

In a standard constraint-based program analysis algorithm, having defined the constraints of interest, the next step would be to give an algorithm for solving them. However, we know of no algorithm for solving equations such as $(*)$. The difficulty is illustrated by the unfolding of $(*)$ given above; because the recursive constraint introduces the equivalent of an unbounded number of quantified variables, it is not obvious how to come up with an equivalent but finite and non-recursive representation that makes explicit all possible solutions.

However, there is a way to side-step this problem entirely. As mentioned above, in practice, clients of a program analysis need to answer either a *may* analysis query (e.g., May `queryUser` return `true`?) or a *must* analysis query (e.g., Must `queryUser` return `true`?). While we do not solve the constraints in general, there is a sound and complete algorithm for deciding may/must queries.

Consider the may analysis query: Is it possible for `queryUser` to return `true`, in other words, is constraint $(*)$ satisfiable? Our algorithm decides this question as follows. First, for a constraint C, we compute a modified constraint $\lceil C \rceil$ that is a *necessary condition* for C. By definition, $\lceil C \rceil$ is a necessary condition for C if $C \Rightarrow \lceil C \rceil$. Thus, if $\lceil C \rceil$ is not identically `false`, then the property may hold; if $\lceil C \rceil$ is `false`, then C is never `true`. Now, any choice of necessary condition is sound (if the necessary condition is unsatisfiable then so is the original constraint), but to guarantee completeness (if the necessary condition is satisfiable then so is the original constraint) and termination of our algorithm the necessary condition must satisfy two additional properties:

- For completeness, the necessary condition must be the best possible. The *strongest* necessary condition is implied by every other necessary condition.
- For termination, the necessary condition should be only over the observable variables. Eliminating the unobservable variables is what makes solving the constraints possible; it turns out that the strongest necessary condition on observable variables is still sufficient for completeness.

Consider constraint $(*)$ again. The strongest observable necessary condition for `queryUser` to return `true` is $\alpha = \texttt{true}$; i.e., if the input to `queryUser` is `true`, then it may return `true`, otherwise it cannot return `true`. The computation of necessary conditions for answering may queries, and dually sufficient conditions for answering must queries, is discussed in Sections 5 and 6.

We have implemented our algorithm and experimented with several large open source C applications (see Section 8). We show that for these applications we can answer all may and must queries for two significant applications: computing the path- and context-sensitive condition under which every pointer dereference occurs, and finding null dereference errors. For the former, perhaps our most interesting result is that the observable necessary and sufficient conditions we compute do not grow with program size, showing that our approach should scale to even larger programs. For the latter, we show that the interprocedurally path-sensitive analysis reduces false positives by almost an order of magnitude compared to the intraprocedurally path-sensitive, but interprocedurally path-

insensitive analysis. To summarize, this paper makes the following contributions:

- We distinguish observable and unobservable variables in path- and context-sensitive conditions, and it is this distinction that ultimately allows us to give a sound and complete algorithm.
- We show how to obtain strongest necessary and weakest sufficient observable conditions by eliminating unobservable variables and solving the resulting recursive system of constraints over observable variables.
- We give the first scalable, sound, and complete algorithm for computing a large class of precise path- and context-sensitive program properties.

2. Related Work

In this section we survey previous approaches to path- and context-sensitive analysis. The earliest path-sensitive techniques were developed for explicit state model-checking, where essentially every path through the program is symbolically executed and checked for correctness one at a time. In practice, this approach is used to verify relatively small finite state systems, such as hardware protocols [9].

More recent software model-checking techniques address sound and complete path- and context-sensitive analysis [5, 3, 13]. Building on techniques proposed for context-sensitivity [20, 22], Ball et al. propose Bebop, a whole-program model checking tool for boolean programs [5, 3]. Bebop is similar to our approach in that it exploits the scope of local variables through implicit existential quantification and also deals with recursion through context-free reachability. However, Bebop combines these two steps, while our approach separates them: we first explicitly construct formulas with existentially quantified unobservable variables and then subsequently perform a reachability analysis as a fixed point computation. This design allows us to insert a new step in between that manipulates the existentially quantified formulas, in particular to convert them to (normally) much smaller formulas that preserve may or must queries prior to performing the global reachability computation. This extra step is, we believe, the reason that we are able to scale our approach to programs much larger than have been previously reported for systems using model checking of boolean programs [5, 3, 13]. Another advantage of this approach is that we can use unobservable variables to model fixed, but unknown, parts of the environment (see discussion in Section 1). Our method is also modular, in contrast to most software model checking systems that require the entire program.

Current state-of-the-art software model-checking tools are based on *counter-example driven predicate abstraction* [4, 16]. Predicate abstraction techniques iteratively refine an initial coarse abstraction until a property of interest is either verified or refuted. Refinement-based approaches may not terminate, as the sequence of progressively more precise abstractions is not guaranteed to converge. Our results show that for a large class of properties the exact path- and context-sensitive conditions can be computed directly without refinement and for much larger programs (millions of lines) than the largest programs to which iterative refinement approaches have been applied (about one hundred thousand lines). We believe our techniques could be profitably incorporated into software model checking systems.

An obstacle to scalability in early predicate abstraction techniques was the number of irrelevant predicates along a path. Craig interpolation [16] allows discovery of locally useful predicates and, furthermore, these predicates only involve predicates in scope at a particular program point. Our approach addresses similar issues in a different way: our technique also explicitly accounts for variable scope, and extracting necessary/sufficient conditions eliminates many predicates irrelevant to the queries we want to decide. Unlike interpolants, our technique does not require counter-example traces, and thus does not require the additional machinery of theorem provers and successive refinement steps.

Some of the most scalable techniques for path- and context-sensitive analysis are either unsound or incomplete. For example, ESP is a light-weight and scalable path-sensitive analysis that tracks branch correlations using the idea that conditional tests resulting in different analysis states should be tracked separately, while branches leading to the same analysis state should be merged [11]. ESP's technique is a heuristic and sometimes fails to compute the best path-sensitive condition. Another example of an incomplete system is F-Soft [17]. F-Soft unrolls recursive functions a fixed number of times, resulting in a loss of precision beyond some predetermined recursion depth of k. In contrast, our approach does not impose any limit on the recursion depth and therefore does not lose completeness for programs with recursion. A final example of an incomplete system is Saturn [1]. While Saturn analyses are generally fully path-sensitive within a single procedure, Saturn has no general mechanism for interprocedural path-sensitivity and published Saturn analyses are either interprocedurally path-insensitive or use heuristics to determine which predicates are important to track across function boundaries [23, 12, 8, 14]. We implement the ideas proposed in this paper in Saturn.

Our technique of computing necessary and sufficient conditions is related to the familiar notion of over- and under-approximations used both in abstract interpretation and model checking. For example, Schmidt [21] proposes the idea of over and under-approximating states in abstract interpretation and presents a proof of soundness and completeness for a class of path-insensitive analysis problems. Many model-checking approaches also incorporate the idea of over- and under-approximating reachable states to obtain a more efficient fixed point computation [6, 10]. Our contribution is to show how to compute precise necessary and sufficient conditions while combining context-sensitivity, path-sensitivity, and recursion.

The idea of computing strongest necessary and weakest sufficient conditions for propositional formulae dates back to Boole's technique of eliminating the middle term [7]. Lin presents efficient algorithms for strongest necessary and weakest sufficient conditions for fragments of first-order logic, but does not explore computing strongest necessary and weakest sufficient conditions for the solution of recursive constraints [18].

In our system, the analysis of a function f may be different for different call-sites even within f's definition, which gives it the expressiveness of context-free reachability (in the language of dataflow analysis) or polymorphic recursion (in the language of type theory). Most polymorphic recursive type inference systems are based on *instantiation constraints* [15]. Our formalization is closer to Mycroft's original work on polymorphic recursion, which represents instantiations directly as substitutions [19].

3. Constraints

We use a small functional language to present our techniques:

$$
\begin{array}{lcl}
\textit{Program } P & ::= & F^+ \\
\textit{Function } F & ::= & \textit{define } f(x) = e \\
\textit{Expression } E & ::= & \textit{true} \mid \textit{false} \mid c_i \mid x \mid f(e) \\
 & & \mid \textit{if } e_1 \textit{ then } e_2 \textit{ else } e_3 \mid \textit{let } x = e_1 \textit{ in } e_2 \\
 & & \mid e_1 = e_2 \mid e_1 \wedge e_2 \mid e_1 \vee e_2 \mid \neg e
\end{array}
$$

Expressions are *true*, *false*, *abstract values* c_i, function arguments x, functional calls, conditional expressions, let bindings and comparisons between two expressions. Boolean-valued expressions can be composed using the standard boolean connectives, $\wedge$, $\vee$, and $\neg$. We model unobservable behavior in the language by references to unbound variables, which are by convention taken

to have a non-deterministic value chosen on function invocation. Thus, any free variables occurring in a function body are unobservable. All other sources of unobservable behavior discussed in Section 1 can be modeled using references to undefined variables.

For simplicity of presentation, we assume that boolean-valued expressions are used only in conditionals, that equality comparisons $e_1 = e_2$ are always between expressions that evaluate to abstract values, and that functions return one of the abstract values c_i. This small language includes two essential features needed to motivate and illustrate our techniques. First, there is an expressive language of predicates used in conditionals, so that path-sensitivity is a non-trivial problem. Second, functions can return any one of a set of values c_i, but this set is finite. Intuitively, the c_i's stand for possible abstract values we are interested in assigning to a program.

The goal of our technique is to assign each function constraints of the following form:

DEFINITION 1 (*Constraints).*

$$\begin{array}{lll} \text{Equation } \mathcal{E} & ::= & [\vec{\Pi_i}] = \exists\beta_1,\ldots,\beta_m.[\vec{\mathcal{F}_i}] \\ \text{Constraint } \mathcal{F} & ::= & (\tau_1 = \tau_2) \mid \Pi[C_i/\alpha] \\ & & \mid \mathcal{F}_1 \wedge \mathcal{F}_2 \mid \mathcal{F}_1 \vee \mathcal{F}_2 \mid \neg\mathcal{F} \\ \text{Type } \tau & ::= & \alpha \mid C_i \end{array}$$

Constraints are equations between *types* (type variables and abstract values), constraint variables with a substitution, or boolean combinations of constraints. Constraints express the condition under which a function f with input α returns a particular abstract value c; we usually index the corresponding constraint variable $\Pi_{f,\alpha,C}$ for clarity, though we omit the function name if it is clear from context. So, for example, if there are two abstract values c_1 and c_2, the equation

$$[\Pi_{f,\alpha,C_1},\ \Pi_{f,\alpha,C_2}] = [\mathit{true},\ \mathit{false}]$$

describes the function f that always returns $\mathtt{c}_1$, and

$$[\Pi_{f,\alpha,C_1},\ \Pi_{f,\alpha,C_2}] = [\alpha = C_2,\ \alpha = C_1]$$

describes the function f that returns c_1 if its input is c_2 and vice versa. As a final example, the function

$$\mathtt{define\ f(x)} = \mathtt{if(y = c_2)\ then\ c_1\ else\ c_2}$$

where y is free is modeled by the equation:

$$[\Pi_{f,\alpha,C_1},\ \Pi_{f,\alpha,C_2}] = \exists\beta.[\beta = C_2,\ \beta = C_1]$$

The existentially quantified variable β models the unknown result of referencing y. Note that β is shared by the two constraints; in particular, in any solution β must be either C_1 or C_2, capturing that a function call returns only one value.

Figure 1 presents most of the constraint inference rules for the small language given above. The remaining rules are omitted for lack of space but are all straightforward analogs of the rules shown. Rules 1-5 prove judgments $A \vdash_{\mathit{true,false}} e : \mathcal{F}$, describing the constraints $\mathcal{F}$ under which an expression e evaluates to *true* or *false* in environment A. Rules 6-11 prove judgments $A \vdash_{c_i} e : \mathcal{F}$ that give the constraint under which expression e evaluates to abstract value c_i. Finally, rule 12 constructs systems of equations, giving the (possibly) mutually recursive conditions under which a function returns each abstract value.

We briefly explain a subset of the rules. In Rule 3, two expressions e_1 and e_2 are equal whenever both evaluate to the same value. Rule 8 says that if under environment A, variable x has type α, then x evaluates to c_i only if $\alpha = C_i$. Rule 11 presents the rule for function calls: If the input to function f is the abstract value c_k, and the constraint under which *f* returns c_i is Π_{f,α,C_i}, then *f*(e) has type C_i under the constraint $\mathcal{F}_k \wedge \Pi_{f,\alpha,C_i}[C_k/\alpha]$.

EXAMPLE 1. Suppose we analyze the following function:

$$\mathtt{define\ f(x)} = \mathtt{if((x = c_1) \vee (y = c_2))\ then\ c_1\ else\ f(c_1)}$$

(1) $$\frac{}{A \vdash_{\mathit{true}} \mathit{true} : \mathit{true}}$$

(2) $$\frac{}{A \vdash_{\mathit{true}} \mathit{false} : \mathit{false}}$$

(3) $$\frac{A \vdash_{c_i} e_1 : \mathcal{F}_{1,i} \quad A \vdash_{c_i} e_2 : \mathcal{F}_{2,i}}{A \vdash_{\mathit{true}} (e_1 = e_2) : \bigvee_i (\mathcal{F}_{1,i} \wedge \mathcal{F}_{2,i})}$$

(4) $$\frac{A \vdash_{\mathit{true}} e : \mathcal{F}}{A \vdash_{\mathit{false}} e : \neg\mathcal{F}}$$

(5) $$\frac{A \vdash_{\mathit{true}} e_1 : \mathcal{F}_1 \quad A \vdash_{\mathit{true}} e_2 : \mathcal{F}_2 \quad \otimes \in \{\wedge, \vee\}}{A \vdash_{\mathit{true}} e_1 \otimes e_2 : \mathcal{F}_1 \otimes \mathcal{F}_2}$$

(6) $$\frac{}{A \vdash_{c_i} c_i : \mathit{true}}$$

(7) $$\frac{i \neq j}{A \vdash_{c_i} c_j : \mathit{false}}$$

(8) $$\frac{A(x) = \alpha}{A \vdash_{c_i} x : (\alpha = C_i)}$$

(9) $$\frac{A \vdash_{\mathit{true}} e_1 : \mathcal{F}_1 \quad A \vdash_{c_i} e_2 : \mathcal{F}_2 \quad A \vdash_{c_i} e_3 : \mathcal{F}_3}{A \vdash_{c_i} \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 : (\mathcal{F}_1 \wedge \mathcal{F}_2) \vee (\neg\mathcal{F}_1 \wedge \mathcal{F}_3)}$$

(10) $$\frac{A \vdash_{c_j} e_1 : \mathcal{F}_{1j} \quad A, x : \alpha \vdash_{c_i} e_2 : \mathcal{F}_{2i} \quad (\alpha\ \mathit{fresh})}{A \vdash_{c_i} \mathit{let}\ x = e_1\ \mathit{in}\ e_2 : \bigvee_j (\mathcal{F}_{1j} \wedge \mathcal{F}_{2i} \wedge (\alpha = C_j))}$$

(11) $$\frac{A \vdash_{c_k} e : \mathcal{F}_k}{A \vdash_{c_i} f(e) : \bigvee_k (\mathcal{F}_k \wedge \Pi_{f,\alpha,c_i}[C_k/\alpha])}$$

(12) $$\frac{\alpha \notin \{\beta_1,\ldots,\beta_m\} \quad x : \alpha, y_1 : \beta_1, \ldots, y_n : \beta_m \vdash_{c_i} e : \mathcal{F}_i \quad 1 \leq i \leq n}{\vdash \mathit{define\ f(x)} = e : [\vec{\Pi}_{f,\alpha,c_i}] = \exists\beta_1,\ldots,\beta_m.\ [\vec{\mathcal{F}_i}]}$$

Figure 1. Inference Rules

where y is undefined and the only abstract values are $\mathtt{c}_1$ and $\mathtt{c}_2$. Then

$$\begin{bmatrix} \Pi_{f,\alpha,C_1} \\ \cdots \end{bmatrix} = \exists\beta. \begin{bmatrix} (\alpha = C_1 \vee \beta = C_2) \vee \\ \neg(\alpha = C_1 \vee \beta = C_2) \wedge \Pi_{f,\alpha,C_1}[C_1/\alpha] \\ \cdots \end{bmatrix}$$

is the equation computed by the inference rules (see Figure 2). Note that the substitution $[C_1/\alpha]$ in the formula expresses that the argument of the recursive call to *f* is c_1.

Due to space constraints we can only sketch the semantics of constraints. Constraints are interpreted over the standard four point lattice with $\bot \leq \mathit{true}, \mathit{false}, \top$ and $\bot, \mathit{true}, \mathit{false} \leq \top$, where $\wedge$ is meet, $\vee$ is join, and $\neg\bot{=}\bot$, $\neg\top = \top$, $\neg\mathit{true} = \mathit{false}$, and $\neg\mathit{false} = \mathit{true}$. Given an assignment θ for the existential variables, the meaning of a system of equations E is a standard limit of a series of approximations $\theta(E^0), \theta(E^1), \ldots$ generated by repeatedly unfolding E. We are interested in both the least fixed point (where the first approximation of all Π variables is $\bot$) and greatest fixed point (where the first approximation is $\top$)

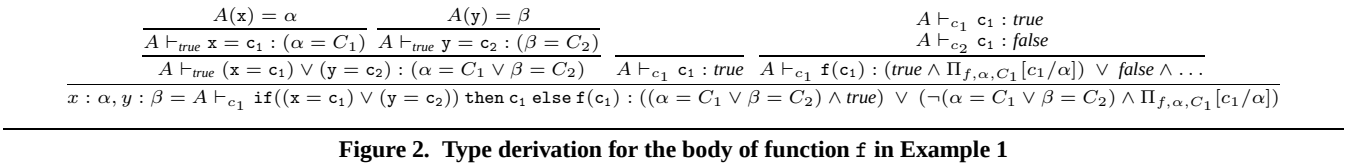

Figure 2. Type derivation for the body of function `f` in Example 1

semantics. The value $\bot$ in the least fixed point semantics (resp. $\top$ in the greatest fixed point) represents non-termination of the analyzed program. We do not attempt to reason about termination, and our results are generally qualified by an assumption that the program terminates. By construction, the inference rules in Figure 1 guarantee that $\Pi_{f,\alpha,C_i} \wedge \Pi_{f,\alpha,C_j} \leq \mathit{false}$ in the least fixed point semantics if $i \neq j$, and $\bigvee_i \Pi_{f,\alpha,C_i} \geq \mathit{true}$ in the greatest fixed point semantics; we rely on this property in Section 6.2.

4. Boolean Constraints

Our main technical result is a sound and complete method for answering satisfiability (may) and validity (must) queries for the constraints of Definition 1. The algorithm has four major steps:

- eliminate the existentially bound (unobservable) variables by extracting necessary/sufficient conditions from the equations;
- rewrite the equations to be monotonic in the Π variables;
- eliminate recursion by a fixed point computation;
- finally, apply a decision procedure to the closed-form equations.

Our target decision procedure for the last step is SAT, and so at some point we must translate our type constraints into equivalent boolean constraints. We perform this translation first, before performing any of the steps above.

For every type variable (observable or unobservable) σ_i, we introduce boolean variables $\sigma_{i1}, ..., \sigma_{in}$ such that σ_{ij} is *true* if and only if $\sigma_i = C_j$. We refer to boolean variables α_{ij} as *observable variables* and β_{ij} as *unobservable variables*. We map the equation variables Π_{f,α,C_i} to boolean variables of the same name. A variable Π_{f,α,C_i} represents the condition under which *f* returns c_i, hence we refer to Π_{f,α,C_i}'s as *return variables*. We also translate each $\tau_1 = \tau_2$ occurring in a type constraint:

$$\begin{array}{lcll} C_i = C_i & \Leftrightarrow & \mathit{true} & \\ C_i = C_j & \Leftrightarrow & \mathit{false} & i \neq j \\ v_i = C_j & \Leftrightarrow & v_{ij} & \end{array}$$

Note that subexpressions of the form $v_i = v_j$ never appear in the constraints generated by the system of Figure 1. We replace every substitution $[C_j/\alpha_i]$ by the boolean substitution $[\mathit{true}/\alpha_{ij}]$ and $[\mathit{false}/\alpha_{ik}]$ for $j \neq k$.

EXAMPLE 2. The first row of Example 1 results in the following boolean constraints (here boolean variable α_1 represents the equation $\alpha = C_1$ and β_2 represents $\beta = C_2$):

$$\Pi_{f,\alpha,C_1} = \exists\beta_2.(\alpha_1 \vee \beta_2) \vee (\neg(\alpha_1 \vee \beta_2) \wedge \Pi_{f,\alpha,C_1}[\mathit{true}/\alpha_1])$$

The existentially quantified variable β_1 and substitution $[\mathit{false}/\alpha_2]$ are omitted because neither β_1 nor α_2 occurs in the formula.

In the general case, the original type constraints result in a recursive system of boolean constraints of the following form:

EQUATION 1.

$$E = \left[\begin{array}{lcl} [\vec{\Pi}_{f_1,\alpha,C_i}] & = & \exists\vec{\beta}_1.[\vec{\phi}_{1i}(\vec{\alpha}_1, \vec{\beta}_1, \vec{\Pi}[\vec{b}_1/\vec{\alpha}])] \\ \vdots & & \vdots \\ [\vec{\Pi}_{f_k,\alpha,C_i}] & = & \exists\vec{\beta}_k.[\vec{\phi}_{ki}(\vec{\alpha}_k, \vec{\beta}_k, \vec{\Pi}[\vec{b}_k/\vec{\alpha}])] \end{array}\right]$$

where $\vec{\Pi} = \langle \Pi_{f_1,\alpha,C_1}, ..., \Pi_{f_k,\alpha,C_n} \rangle$ and $b_i \in \{\mathit{true}, \mathit{false}\}$ and the ϕ's are quantifier-free formulas over $\vec{\beta}$, $\vec{\alpha}$, and $\vec{\Pi}$. The substitutions of the form $\vec{\Pi}[\vec{b}/\vec{\alpha}]$ result from translation of constraints produced by Rule 11 of Figure 1.

4.1 Satisfiability, Validity, and Monotonicity

In this subsection we give a few definitions and technical lemmas used in Section 6 to prove our main result. As it stands the boolean constraints do not quite preserve solutions of the type constraints. We add additional constraints guaranteeing that a solution of the boolean translation of a type constraint guarantees every type variable is assigned some abstract value (*existence*) and that no type variable is assigned multiple abstract values (*uniqueness*):

1. *Uniqueness:* $\psi_{\mathit{unique}} = (\bigwedge_{j \neq k} \neg(v_{ij} \wedge v_{ik}))$
2. *Existence:* $\psi_{\mathit{exist}} = (\bigvee_j v_{ij})$

where v_{ij} is any boolean variable. We can now formulate definitions of satisfiability and validity for our system, *SAT** and *VALID**:

DEFINITION 2. $\mathit{SAT}^*(\phi) \equiv \mathit{SAT}(\phi \wedge \psi_{\mathit{exist}} \wedge \psi_{\mathit{unique}})$

In other words, any satisfying assignment must observe the existence and uniqueness assumptions for all boolean variables v_{ij}.

DEFINITION 3. $\mathit{VALID}^*(\phi) \equiv (\{\psi_{\mathit{exist}}\} \cup \{\psi_{\mathit{unique}}\} \models \phi)$

Using *VALID**, we can conclude, for example, that $\alpha_{11} \vee \alpha_{12}$ is a tautology in a language with two abstract values $\mathtt{c}_1$ and $\mathtt{c}_2$.

DEFINITION 4. Let ϕ be a quantifier-free formula over α_{ij}, β_{ij}, and Π_{ij}. Let $\mathcal{M}(\phi)$ be ϕ converted to negation normal form (negations driven in and $\neg\neg x$ replaced by x) and replacing any negative literal $\neg v_{ij}$ by $\bigvee_{k \neq j} v_{ik}$. Then $\mathcal{M}(\phi)$ is *monotonic in* v_{ij}.

LEMMA 1. $\mathit{SAT}^*(\phi) \Leftrightarrow \mathit{SAT}(\mathcal{M}(\phi) \wedge \psi_{\mathit{unique}})$

PROOF. Consider any satisfying assignment $\bar{v}$ to $\mathcal{M}(\phi) \wedge \psi_{\mathit{unique}}$. Suppose that $\bar{v}$ assigns all v_{ij} to *false* such that ψ_{exist} is violated. But since $\mathcal{M}(\phi)$ does not contain any negations, setting one v_{ij} to *true* satisfies $\mathcal{M}(\phi)$, ψ_{exist}, and ψ_{unique}, implying $\mathit{SAT}^*(\phi)$. The other direction is also easy and omitted. □

LEMMA 2. $\mathit{VALID}^*(\phi) \Leftrightarrow (\{\psi_{\mathit{exist}}\} \models \mathcal{M}(\phi))$

PROOF. Dual to the proof of Lemma 1. □

The original constraints have four possible meanings (c.f., Section 3) while the boolean constraints have only two. We claim without proof that the translation is correct in that whenever the meaning of the original constraints is either *true* or *false* (i.e., the original program terminates), the translation has the same meaning.

5. Necessary and Sufficient Conditions

As discussed in previous sections, a key step in our algorithm is extracting necessary/sufficient conditions from a system of constraints C. The necessary (resp. sufficient) conditions should be satisfiable (resp. valid) if and only if C is satisfiable (resp. valid). This section makes precise exactly what necessary/sufficient conditions we need; in particular, there are two technical requirements:

- The necessary (resp. sufficient) conditions should be as *strong* (resp. *weak*) as possible.
- The necessary/sufficient conditions should be only over observable variables.

In the following, we use $\mathcal{V}^-(\phi)$ (resp. $\mathcal{V}^+(\phi)$) to denote the set of unobservable (resp. observable) variables β_{ij} (resp. α_{ij}) used in ϕ.

DEFINITION 5. Let ϕ be a quantifier-free formula. We say $\lceil\phi\rceil$ is the *strongest observable necessary condition* for ϕ if:

$$\begin{array}{ll}(1) & \phi \Rightarrow \lceil\phi\rceil \\ (2) & \forall\phi'.((\phi \Rightarrow \phi') \Rightarrow (\lceil\phi\rceil \Rightarrow \phi')) \\ & \textit{where } \mathcal{V}^-(\phi') = \emptyset \ \wedge \ \mathcal{V}^+(\phi') = \mathcal{V}^+(\phi)\end{array}$$

The first condition says $\lceil\phi\rceil$ is necessary for ϕ, and the second condition ensures $\lceil\phi\rceil$ is stronger than any other necessary condition with respect to ϕ's observable variables $\mathcal{V}^+(\phi)$. The additional restriction $\mathcal{V}^-(\lceil\phi\rceil) = \emptyset$ enforces that the strongest necessary condition for a formula ϕ has no unobservable variables.

DEFINITION 6. Let ϕ be a quantifier-free formula. We say $\lfloor\phi\rfloor$ is the *weakest observable sufficient condition* for ϕ if:

$$\begin{array}{ll}(1) & \lfloor\phi\rfloor \Rightarrow \phi \\ (2) & \forall\phi'.((\phi' \Rightarrow \phi) \Rightarrow (\phi' \Rightarrow \lfloor\phi\rfloor)) \\ & \textit{where } \mathcal{V}^-(\phi') = \emptyset \ \wedge \ \mathcal{V}^+(\phi') = \mathcal{V}^+(\phi)\end{array}$$

We include the following variant of well-known results.

LEMMA 3. Observable strongest necessary and weakest sufficient conditions for any formula ϕ exist and are unique up to logical equivalence.

Strongest necessary and weakest sufficient conditions are immediately useful in program analysis for answering queries about program properties. For example, let ϕ be the condition under which a given program property P holds. It follows immediately from the definition of necessary and sufficient conditions that:

- If *SAT*($\lceil\phi\rceil$), then P *MAY* hold.
- If *VALID*($\lfloor\phi\rfloor$), then P *MUST* hold.

Furthermore, for the strongest and weakest such conditions, we have the following additional guarantees:

- If *UNSAT*($\lceil\phi\rceil$), then P *MUST NOT* hold.
- If *INVALID*($\lfloor\phi\rfloor$), then P *MAY NOT* hold.

In this sense, strongest necessary and weakest sufficient conditions of ϕ define a tight observable bound on ϕ. If ϕ has only observable variables, then the strongest necessary and weakest sufficient conditions of ϕ are equivalent to ϕ. If ϕ has only unobservable variables, then the best possible bounds are $\lceil\phi\rceil = \textit{true}$ and $\lfloor\phi\rfloor = \textit{false}$. Intuitively, the "difference" between strongest necessary and weakest sufficient conditions of a formula define the amount of uncertainty present in the original formula.

EXAMPLE 3. Suppose we are interested in determining the conditions under which a pointer is dereferenced in a function call. Consider the implementations of `f` and `g` given in Figure 3.

Scanning the implementations of `f` and `g`, we see that `p` is dereferenced under the following constraint:

$$\texttt{p!=NULL} \wedge \texttt{flag!=0} \wedge \texttt{buf!=NULL} \wedge *\texttt{buf} =='\ \texttt{i}'$$

Since the return value of `malloc` (i.e., `buf`) and the user input (i.e., $*$`buf`) are unobservable outside of `f`, the strongest observable necessary condition for `f` to dereference `p` is given by the simpler condition:

$$\texttt{p!=NULL} \ \wedge \ \texttt{flag!=0}$$

```
 1. int g(int* p) {
 2.   if(p==NULL) return -1;
 3.   return 1;
 4. }
 5. void f(int* p, int flag) {
 6.  if(g(p)<0 || !flag) return;
 7.  char* buf = malloc(sizeof(char));
 8.  if(!buf) return;
 9.  *buf = getUserInput();
10.  if(*buf=='i')
11.   *p = 1;
12. }
```

Figure 3. Example code.

On the other hand, nothing in a calling context of `f` guarantees that `p` is dereferenced when `f` is called; hence, the weakest observable sufficient condition for the dereference is `false`.

6. Solving the Constraints

In this section, we give an algorithm for computing observable strongest necessary and weakest sufficient conditions for the equations given in Section 4. Our algorithm first eliminates existentially quantified variables from every formula (Section 6.1). We then transform the equations to both be *monotonic* in the return variables and preserve strongest necessary (weakest sufficient) conditions under substitution (Section 6.2). Finally, we solve the equations to eliminate recursive constraints (Section 6.3), yielding a system of (non-recursive) formulas over observable variables. Each step preserves the satisfiability/validity of the original equations, and thus the original may/must query can be decided using a standard SAT solver on the final formulas.

6.1 Eliminating Unobservable Variables

The first step of our algorithm is to eliminate each existentially quantified unobservable variable β_{ij}. We use the following result:

LEMMA 4.

1. The strongest necessary condition of ϕ not containing v is:
$$\textit{SNC}(\phi, v) \equiv \phi[\textit{true}/v] \vee \phi[\textit{false}/v]$$
2. The weakest sufficient condition of ϕ not containing v is:
$$\textit{WSC}(\phi, v) \equiv \phi[\textit{true}/v] \wedge \phi[\textit{false}/v]$$

Proofs of these results were first given by Boole [7]. This technique for computing strongest necessary and weakest sufficient conditions for formulas not containing a given variable v_i is sometimes referred to as *eliminating the middle term* or *forgetting* a variable.

Recall from Section 4.1 that any satisfiable formula must also satisfy existence and uniqueness constraints, while any formula entailed by $\psi_{\textit{exist}}$ and $\psi_{\textit{unique}}$ is a tautology. For example, the strongest necessary condition for $\beta_{11} \wedge \beta_{12}$ not containing β's is *false* in our system, even though applying the technique from Lemma 4 yields *true*. Similar problems arise for weakest sufficient conditions. To compute strongest necessary and weakest sufficient conditions that obey the additional existence and uniqueness conditions of our system, we define *SNC** and *WSC** as follows:

DEFINITION 7.

1. The strongest necessary condition *SNC** of ϕ without v is:
$$\begin{array}{rcl}\textit{SNC}^*(\phi, v) & \equiv & (\phi \wedge \psi_{\textit{exist}} \wedge \psi_{\textit{unique}})[\textit{true}/v] \vee \\ & & (\phi \wedge \psi_{\textit{exist}} \wedge \psi_{\textit{unique}})[\textit{false}/v]\end{array}$$

2. The weakest sufficient condition WSC^* of ϕ without v is:

$$WSC^*(\phi, v) \equiv (\phi \vee \neg\psi_{exist} \vee \neg\psi_{unique})[true/v] \wedge (\phi \vee \neg\psi_{exist} \vee \neg\psi_{unique})[false/v]$$

That these are in fact the strongest necessary/weakest sufficient observable conditions follows from Lemma 4 and Definitions 2 and 3. We compute necessary (resp. sufficient conditions) by replacing all expressions $\exists v.\phi$ by $SNC^*(\phi, v)$ (resp. $WSC^*(\phi, v)$). Thus, this step yields two distinct sets of equations, one for necessary and one for sufficient conditions.

Note that, in the general case of Lemma 4, the strongest necessary and weakest sufficient conditions of any formula may double the size of the original formula. However, it is easy to see that if a literal v occurs only positively in ϕ, the strongest necessary condition can be computed as $\phi[v/true]$, and if v occurs only negatively, the strongest necessary condition is given by $\phi[v/false]$. Analogous optimizations apply to weakest sufficient conditions. Furthermore, it is not always necessary to add the existence and uniqueness constraints for any unobservable variable as suggested by Definition 7. For example, if a formula contains β_{ij}, but no β_{ik}, it is unnecessary to add the explicit uniqueness constraint $\neg(\beta_{ij} \wedge \beta_{ik})$.

EXAMPLE 4. Consider the function given in Example 1, for which boolean constraints are given in Example 2. We compute the strongest necessary condition for Π_{f,α,C_1}:

$$\begin{array}{rcl} \lceil \Pi_{f,\alpha,C_1} \rceil & = & (\alpha_1 \vee true) \vee \\ & & (\neg(\alpha_1 \vee true) \wedge \lceil \Pi_{f,\alpha,C_1} \rceil [true/\alpha_1]) \\ & \vee & (\alpha_1 \vee false) \vee \\ & & (\neg(\alpha_1 \vee false) \wedge \lceil \Pi_{f,\alpha,C_1} \rceil [true/\alpha_1]) \\ & = & true \end{array}$$

The reader can verify that the weakest sufficient condition for Π_{f,α,C_1} is also *true*. In the above derivation, the existence and uniqueness constraints are omitted since they are redundant.

After eliminating unobservable variables from formulas ϕ_{ij} of Equation 1, we obtain two systems of constraints E_{NC} and E_{SC}, giving the strongest necessary and weakest sufficient observable conditions, respectively:

EQUATION 2.

$$E_{\text{NC}} = \left[\begin{array}{rcl} \lceil \Pi_{f_1,\alpha,C_1} \rceil & = & \phi'_{11}(\vec{\alpha_1}, \lceil \vec{\Pi} \rceil [\vec{b}_1/\vec{\alpha}]) \\ & \vdots & \\ \lceil \Pi_{f_k,\alpha,C_n} \rceil & = & \phi'_{kn}(\vec{\alpha}_k, \lceil \vec{\Pi} \rceil [\vec{b}_k/\vec{\alpha}]) \end{array} \right]$$

E_{SC} is analogous to E_{NC}.

6.2 Preservation Under Substitution

Our goal is to solve the recursive system given in Equation 2 by an iterative, fixed point computation. However, there is a problem: as it stands, Equation 2 may not preserve strongest necessary and weakest sufficient conditions under substitution, a serious problem if we are to compute fixed points by repeated substitution.

EXAMPLE 5.

```
define g(x) =  if(x = c1 ∧ z = c2) then c1 else c2
define f(x) =  let y = g(c1) in
               if(¬(y = c1)) then c1 else c2
```

Suppose we want the strongest necessary condition for `f` returning $\mathtt{c}_1$. Using the machinery presented so far, we compute:

$$\lceil \Pi_{f,\alpha,C_1} \rceil = \neg(\lceil \Pi_{g,\alpha,C_1} \rceil [true/\alpha_1][false/\alpha_2])$$
$$\lceil \Pi_{g,\alpha,C_1} \rceil = \alpha_1$$

where α_1 represents the constraint $\alpha = C_1$, and α_2 represents $\alpha = C_2$. When we replace α_1 for the occurrence of $\lceil \Pi_{g,\alpha,C_1} \rceil$ in the first equation, we obtain *false* as the strongest necessary condition for `f` to return $\mathtt{c}_1$. This result is wrong, since `f` returns $\mathtt{c}_1$ if the variable `z` in function `g` is $\mathtt{c}_1$. Thus, the strongest necessary condition for `f` returning $\mathtt{c}_1$ is *true*.

This example illustrates that Equation 2 does not preserve strongest necessary conditions under substitution; in fact, the result we obtained is not even a necessary condition. The problem arises because $\lceil \neg\phi \rceil \neq \neg\lceil \phi \rceil$. To ensure that strongest necessary and weakest sufficient conditions are preserved under substitution, the return variables may only occur monotonically in a formula.

Note that replacing $\neg\Pi_{f,\alpha,C_i}$ by $\bigvee_{j \neq i} \Pi_{f,\alpha,C_j}$ is not sufficient to solve the problem, because the satisfiability of a formula in our system also requires that the formula obey the uniqueness constraint $\neg(\bigwedge_{i \neq j} \neg(\Pi_{f,\alpha,C_i} \wedge \Pi_{f,\alpha,C_j}))$ which also contains negations on return variables. Similar problems arise for weakest sufficient conditions because the existence constraint $\bigvee_j \Pi_{f,\alpha,C_j}$ appears anti-monotonically in the definition of validity.[1]

Fortunately, we can transform E_{NC} and E_{SC} into monotonic system of equations $\mathcal{T}(E_{NC})$ and $\mathcal{T}(E_{SC})$ such that:

1. The latter equations contain no negations on return variables.
2. $SAT^*(E_{NC}) \Leftrightarrow SAT(\mathcal{T}(E_{NC}))$
3. $VALID^*(E_{SC}) \Leftrightarrow VALID(\mathcal{T}(E_{SC}))$

The first property is necessary to guarantee that strongest necessary and weakest sufficient conditions are preserved under substitution, while conditions 2 and 3 are required to ensure that strongest necessary and weakest sufficient conditions obtained by our algorithm are satisfiability and validity preserving respectively.

Lemma 5 below states that $\mathcal{T}(E_{NC})$ has properties 1 and 2 listed above, and the proof of the lemma presents an outline of this transformation. Lemma 6 states that $\mathcal{T}(E_{NC})$ preserves strongest necessary conditions under syntactic substitution such that the strongest necessary conditions computed by our technique are satisfiability preserving. Lemmas 7 and 8 state dual results for weakest sufficient conditions and validity preservation.

LEMMA 5. For a system of equations E_{NC}, there exists a system $\mathcal{T}(E_{NC})$ in which all return variables occur only monotonically and for $\phi \in E_{NC}$ and $\phi' \in \mathcal{T}(E_{NC})$, $SAT^*(\phi) \Leftrightarrow SAT(\phi')$.

PROOF. From Lemma 1, we have $SAT^*(\phi) \Leftrightarrow SAT(\mathcal{M}(\phi) \wedge \psi_{unique})$. Hence it suffices to show there exists a ϕ' such that:

$$SAT(\phi') \Leftrightarrow SAT(\mathcal{M}(\phi) \wedge \psi_{unique})$$

To obtain ϕ', we convert $\mathcal{M}(\phi)$ to disjunctive normal form:

$$\phi' = DNF(\mathcal{M}(\phi)) = \begin{array}{l} (p_{11} ... \wedge ... p_{1n}) \vee ... (p_{i1} \wedge ... p_{ij} ... \wedge p_{in}) \\ \vee ... (p_{m1} ... \wedge ... p_{mn}) \end{array}$$

We enforce uniqueness by dropping contradictions from every disjunct of the form $(p_{i1} \wedge ... p_{ij} ... \wedge p_{in})$, i.e., if any disjunct contains both Π_{f,α,C_i} and Π_{f,α,C_j} for $i \neq j$, we replace the clause by *false*. The final formula ϕ' is equi-satisfiable to $\mathcal{M}(\phi) \wedge \psi_{unique}$. □

Now we can show that $\mathcal{T}(E_{NC})$ preserves strongest necessary conditions under substitution. This result is what ultimately guarantees our algorithm computes the strongest necessary condition for the original recursive system given in Equation 1.

[1] The uniqueness and existence constraints of the form $\neg(\bigwedge_{i \neq j} \neg(\Pi_{f,\alpha,C_i} \wedge \Pi_{f,\alpha,C_j}))$ and $\bigvee_j \Pi_{f,\alpha,C_j}$ only apply to Π variables arising from the same call site. While we do not explicitly label Π variables with their respective instantiation sites, from here on, we assume that the uniqueness and existence constraints only apply to return variables arising from the same call site. We rely on this assumption in the proof of Lemmas 5 and 7.

LEMMA 6. Let $\phi \in \mathcal{T}(E_{NC})$ be a formula containing literals $\vec{\alpha}$ and $\vec{\Pi}$. Let $\mathcal{F}$ be the strongest necessary condition for a return variable Π_{α,C_i} only containing observable variables. Then the strongest necessary condition for ϕ not containing Π_{α,C_i} is given by:

$$\lceil \phi \rceil = \phi[\mathcal{F}/\Pi_{\alpha,C_i}]$$

PROOF. We first show that $\phi[\mathcal{F}/\Pi_{\alpha,C_i}]$ is a necessary condition for ϕ. If $\phi = \Pi_{\alpha,C_i}$, then $\phi[\mathcal{F}/\Pi_{\alpha,C_i}] = \mathcal{F}$. So, $\phi \Rightarrow \phi[\mathcal{F}/\Pi_{\alpha,C_i}]$ since $\Pi_{\alpha,C_i} \Rightarrow \mathcal{F}$. If $\phi = v$ where $v \neq \Pi_{\alpha,C_i}$, then $\phi \Rightarrow \phi[\mathcal{F}/\Pi_{\alpha,C_i}]$ since $\phi[\mathcal{F}/\Pi_{\alpha,C_i}] = \phi$. Note that we do not need to consider the case $\phi = \neg\Pi_{\alpha,C_i}$ since Π_{α,C_i}'s occur only monotonically. The first of two cases for the inductive step is (1) $\phi = \phi_1 \wedge \phi_2$. Suppose $\phi_1 \wedge \phi_2 \not\Rightarrow (\phi_1 \wedge \phi_2)[\mathcal{F}/\Pi_{\alpha,C_i}]$ such that $\phi_1 \wedge \phi_2 \not\Rightarrow (\phi_1[\mathcal{F}/\Pi_{\alpha,C_i}]) \wedge (\phi_2[\mathcal{F}/\Pi_{\alpha,C_i}])$ $(*)$
Then, there is a truth assignment $\bar{v}$ satisfying $\phi_1 \wedge \phi_2$, but not $(\phi_1[\mathcal{F}/\Pi_{\alpha,C_i}])) \wedge (\phi_2[\mathcal{F}/\Pi_{\alpha,C_i}])$. By induction,

$$\phi_1 \Rightarrow \phi_1[\mathcal{F}/\Pi_{\alpha,C_i}] \ \wedge \ \phi_2 \Rightarrow \phi_2[\mathcal{F}/\Pi_{\alpha,C_i}]$$

Hence if ϕ_1 and ϕ_2 are *true* then $(\phi_1[\mathcal{F}/\Pi_{\alpha,C_i}]))\wedge(\phi_2[\mathcal{F}/\Pi_{\alpha,C_i}])$ must also be *true*, yielding a contradiction with $(*)$. Case (2) $\phi = \phi_1 \vee \phi_2$ is similar to case (1).

We now show $\phi[\mathcal{F}/\Pi_{\alpha,C_i}]$ is the strongest necessary condition for ϕ, i.e., $\phi[\mathcal{F}/\Pi_{\alpha,C_i}] \Rightarrow \phi[\mathcal{F}''/\Pi_{\alpha,C_i}]$ for any necessary condition $\mathcal{F}''$. Consider the case where $\phi = \Pi_{\alpha,C_i}$. Let $\phi[\mathcal{F}''/\Pi_{\alpha,C_i}]$ be another necessary condition for ϕ. Since $\mathcal{F}$ is the strongest necessary condition for Π_{α,C_i}, we have $\mathcal{F} \Rightarrow \mathcal{F}''$ and hence $\phi[\mathcal{F}/\Pi_{\alpha,C_i}] \Rightarrow \phi[\mathcal{F}''/\Pi_{\alpha,C_i}]$. The case where $\phi = v$ and $v \neq \Pi_{\alpha,C_i}$ is also trivially true.

There are two inductive cases. The first is (1) $\phi = \phi_1 \wedge \phi_2$. Let $\phi[\mathcal{F}''/\Pi_{\alpha,C_i}]$ be another necessary condition for ϕ such that

$$\phi[\mathcal{F}/\Pi_{\alpha,C_i}] \not\Rightarrow \phi[\mathcal{F}''/\Pi_{\alpha,C_i}]$$

Then, there must exist a truth assignment $\bar{v}$ satisfying $\phi[\mathcal{F}/\Pi_{\alpha,C_i}]$, but not $\phi[\mathcal{F}''/\Pi_{\alpha,C_i}]$. By induction:

$$\phi_1[\mathcal{F}/\Pi_{\alpha,C_i}] \Rightarrow \phi_1[\mathcal{F}''/\Pi_{\alpha,C_i}] \ \wedge$$
$$\phi_2[\mathcal{F}/\Pi_{\alpha,C_i}] \Rightarrow \phi_2[\mathcal{F}''/\Pi_{\alpha,C_i}]$$

Hence,

$$(\phi_1[\mathcal{F}/\Pi_{\alpha,C_i}]\wedge\phi_2[\mathcal{F}/\Pi_{\alpha,C_i}]) \Rightarrow (\phi_1[\mathcal{F}''/\Pi_{\alpha,C_i}]\wedge\phi_2[\mathcal{F}''/\Pi_{\alpha,C_i}])$$

thus $\bar{v}$ must also satisfy $\phi[\mathcal{F}''/\Pi_{\alpha,C_i}]$, a contradiction. Case (2) $\phi = \phi_1 \vee \phi_2$ is again symmetric to case (1). □

The dual of this result holds for weakest sufficient conditions.

LEMMA 7. For every system of equations E_{SC}, there exists another system of equations $\mathcal{T}(E_{SC})$ in which all return predicates occur only monotonically and for every $\phi \in E_{SC}$ and $\phi' \in \mathcal{T}(E_{SC})$ such that $\mathit{VALID}^*(\phi) \Leftrightarrow \mathit{VALID}(\phi')$.

PROOF. From Lemma 2, we have $\mathit{VALID}^*(\phi) \Leftrightarrow \{\psi_{\mathit{exist}} \models \mathcal{M}(\phi)\}$. Hence it suffices to show there exists a ϕ' such that $(\{\psi_{\mathit{exist}}\} \models \phi) \Leftrightarrow (\{\} \models \phi')$. We obtain such a ϕ' by converting $\mathcal{M}(\phi)$ to conjunctive normal form:

$$\phi' = \mathit{CNF}(\phi) = \begin{array}{l}(p_{11} ... \vee ... p_{1n}) \wedge ... (p_{i1} \vee ... p_{ij} ... \vee p_{in}) \\ \wedge ... (p_{m1} ... \vee ... p_{mn})\end{array}$$

We then eliminate all tautologies by replacing every clause that contains Π_{α,C_i} for all i, $1 \leq i \leq n$, with *true*. It is easy to see that $(\{\psi_{\mathit{exist}}\} \models \phi) \Leftrightarrow (\{\} \models \phi')$. □

The following lemma states that $\mathcal{T}(E_{SC})$ is validity-preserving under substitution:

LEMMA 8. Let $\phi \in \mathcal{T}(E_{SC})$ be a formula containing literals $\vec{\alpha}$ and $\vec{\Pi}$. Let $\mathcal{F}$ be the weakest sufficient condition for Π_{α,C_i}. Then the weakest sufficient condition for ϕ not containing Π_{α,C_i} is:

$$\lfloor \phi \rfloor = \phi[\mathcal{F}/\Pi_{\alpha,C_i}]$$

PROOF. The proof is similar to the proof of Lemma 6. □

6.3 Eliminating Recursion

After eliminating unobservable variables and transforming the constraints into monotonic satisfiability and validity-preserving systems respectively, we obtain the following systems of equations:

EQUATION 3.

$$\mathcal{T}(E_{NC}) = \left[\begin{array}{l} \mathcal{T}(\lceil \Pi_{f_1,\alpha,C_1} \rceil) = \ \phi_{11}(\vec{\alpha}_1, \mathcal{T}(\lceil \vec{\Pi} \rceil)[\vec{b}_1/\vec{\alpha}_i]) \\ \vdots \\ \mathcal{T}(\lceil \Pi_{f_k,\alpha,C_n} \rceil) = \ \phi_{kn}(\vec{\alpha}_k, \mathcal{T}(\lceil \vec{\Pi} \rceil)[\vec{b}_k/\vec{\alpha}_i]) \end{array}\right]$$

The system $\mathcal{T}(E_{SC})$ is analogous.

Consider vectors of boolean formulas $\vec{\gamma}$ over the α_{ij}'s appearing in the constraints; these formulas have no unobservable or return variables. We define a lattice L with the following ordering:

$$\begin{array}{ll} \bot_{NC} = \overrightarrow{\mathit{false}}^{n \cdot m} & \bot_{SC} = \overrightarrow{\mathit{true}}^{n \cdot m} \\ \top_{NC} = \overrightarrow{\mathit{true}}^{n \cdot m} & \top_{SC} = \overrightarrow{\mathit{false}}^{n \cdot m} \\ \vec{\gamma}_1 \sqcup_{NC} \vec{\gamma}_2 = \langle ..., \gamma_{1i} \vee \gamma_{2i}, ... \rangle & \vec{\gamma}_1 \sqcup_{SC} \vec{\gamma}_2 = \langle ..., \gamma_{1i} \wedge \gamma_{2i}, ... \rangle \end{array}$$

The lattice L is finite (up to logical equivalence) since there are only a finite number of variables α_{ij} and hence only a finite number of logically distinct formulas. We define two functions from L to L

$$F_{NC}(\vec{\gamma}_{NC}) = \langle \ldots, \phi_{ij}[\vec{\gamma}_{NC}/\mathcal{T}(\lceil \vec{\Pi} \rceil)], \ldots \rangle$$
$$F_{SC}(\vec{\gamma}_{SC}) = \langle \ldots, \phi_{ij}[\vec{\gamma}_{SC}/\mathcal{T}(\lfloor \vec{\Pi} \rfloor)], \ldots \rangle$$

substituting boolean formulas $\vec{\gamma}$ for return variables $\mathcal{T}(\vec{\Pi}_{\alpha,C_i})$. We compute a least fixed point solution for E_{NC} as $\mathit{fix}(F_{NC}(\bot_{NC}))$ and for E_{SC} as $\mathit{fix}(F_{SC}(\bot_{SC}))$. The fixed points exist because both systems are monotonic in $\mathcal{T}(\lceil \vec{\Pi} \rceil)$ and $\mathcal{T}(\lfloor \vec{\Pi} \rfloor)$ respectively.

EXAMPLE 6. Recall that in Example 4 we computed $\lfloor \Pi_{f,\alpha,C_1} \rfloor$ for the function `f` defined in Example 1 as:

$$\lfloor \Pi_{f,\alpha,C_1} \rfloor \quad = \quad \alpha_1 \vee (\neg\alpha_1 \wedge \lfloor \Pi_{f,\alpha,C_1} \rfloor[\mathit{true}/\alpha_1])$$

Note that this formula is already monotonic in $\lfloor \Pi_{f,\alpha,C_1} \rfloor$ and does not contain any contradictions or tautologies. To find the weakest sufficient condition for Π_{f,α,C_1}, we first substitute *true* for $\lfloor \Pi_{f,\alpha,C_1} \rfloor$. This yields the formula $\alpha_1 \vee \neg\alpha_1$, a tautology. As a result, our algorithm finds the fixed point solution *true* for the weakest sufficient condition of Π_{f,α,C_1}. Since `f` is always guaranteed to return c_1, the weakest sufficient condition computed using our algorithm is the most precise solution possible.

7. Implementation

We have implemented our method in Saturn, a summary-based, context, and intraprocedurally path-sensitive analysis framework [1]. Our implementation extends the existing Saturn infrastructure to allow client analyses to query fully interprocedural strongest necessary and weakest sufficient conditions for the intraprocedural constraints computed by Saturn, where function return values and side effects are represented as unconstrained variables.[2] For example, given an intraprocedural constraint computed by Saturn, such as `x = 1` $\wedge$ `queryUser(y) = true` for the `queryUser` function from Section 1, our analysis yields the interprocedural constraints

[2] Saturn treats loops as tail-recursive functions; hence, we also compute strongest necessary and weakest sufficient conditions for side effects of loops.

$\texttt{x} = 1 \wedge \texttt{y} = \texttt{true}$ as the strongest necessary condition and `false` as the weakest sufficient condition.

While it is important in our technique that the set of possible values can be exhaustively enumerated (i.e., so that the complement of $\neg\Pi_{\alpha,C_i}$ is expressible as a finite disjunction, recall Section 6.2), it is not necessary that the set be finite, but only finitary, that is, finite for a given program. Furthermore, while it is clear that the technique can be applied to finite state properties or enumerated types, it can also be extended to any property where a finite number of equivalence classes can be derived to describe the possible outcomes. Our implementation goes beyond finite state properties; it first collects the set of all predicates corresponding to comparisons between function return values (and side effects) and constants. For instance, if a condition such as $\texttt{if}(\texttt{foo}(\texttt{a}) == 3)$ is used at some call site of `foo`, then we compute strongest necessary and weakest sufficient conditions for $\Pi_{\texttt{foo,a},3}$ and its negation. This technique allows us to finitize the interesting set of return values associated with a function and makes it possible to use the algorithms described so far with minor modifications. Note that any finitization strategy entails a loss of precision in some situations. For example, if the return values of two arbitrary functions `f` and `g` are compared with each other, the strategy we use may not allow us to determine the exact necessary and sufficient condition under which `f` and `g` return the same value.

The algorithm of Section 6.3 computes a least fixed point. However, the underlying Saturn infrastructure can fail by exceeding resource limits (e.g., time-outs); if any iteration of the fixed point computation failed to complete we would be left with unsound approximations. Thus, our implementation computes a greatest fixed point, as we can halt at any iteration and still have sound results. The greatest fixed point is less precise than the least fixed point in some cases, such as for non-terminating computation paths. For instance, for the simple everywhere non-terminating function:

$$\texttt{define f(x)} = \texttt{if}(\texttt{f(x)} = \texttt{c}_1)\ \texttt{then c}_1\ \texttt{else c}_2$$

the greatest fixed point computation yields *true* for the strongest necessary condition for `f` returning $\texttt{c}_1$ while the least fixed point computation yields *false*.

The toy language used in the technical part of the paper assumes that each function has exactly one output (i.e., functions do not have side effects). This restriction makes it safe to eliminate all unobservable variables while still guaranteeing completeness for finite domains. When functions have multiple outputs, however, there may also be correlations between different outputs, where the correlation is established through the use of an unobservable variable. The example below illustrates such a correlation:

```
int foo(int** p) {
  int* x = malloc(sizeof(int));
  if(!x) return -1;
  *p = x;
  return 1;
}
```

Note that the predicate $\texttt{foo(p)} == 1$ implies that $*\texttt{p}$ is initialized to a non-null value; however, we cannot reason about this correlation if we eliminate the unobservable variable corresponding to the return value of `malloc`. In such cases, our implementation introduces additional variables describing output state to capture externally visible dependencies between different outputs.

8. Experimental Results

We conducted two sets of experiments to evaluate our technique on OpenSSH, Samba, and the Linux kernel. In the first set of experiments we compute necessary and sufficient conditions for pointer dereferences. Pointer dereferences are ubiquitous in C programs and computing the necessary and sufficient conditions for each and every syntactic pointer dereference to execute is a good stress test

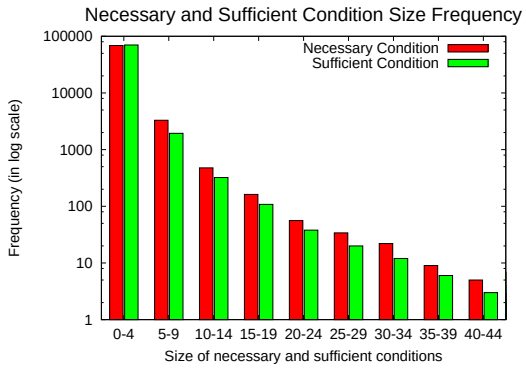

Figure 4. Frequency of necessary and sufficient condition sizes (in terms of the number of boolean connectives) at sinks for Linux

	Linux 2.6.17.1	Samba 3.0.23b	OpenSSH 4.3p2
Average original guard size	3.00	4.45	3.02
Average NC size (sink)	0.75	1.02	0.75
Average SC size (sink)	0.48	0.67	0.50
Average NC size (source)	2.39	2.82	1.39
Average SC size (source)	0.45	0.49	0.67
Average call chain depth	5.98	4.67	2.03
Lines of code	6,275,017	515,689	155,660

Figure 5. Necessary and sufficient condition sizes (in terms of number of boolean connectives in the formula) for pointer dereferences.

for our approach. As a second experiment, we incorporate our technique into a null dereference analysis and demonstrate that our technique reduces the number of false positives by close to an order of magnitude without resorting to ad-hoc heuristics or compromising soundness.

In our first set of experiments, we measure the size of necessary and sufficient conditions for pointer dereferences both at *sinks*, where pointers are dereferenced, and at *sources*, where pointers are first allocated or read from the heap. In Figure 3, consider the pointer dereference (sink) at line 11. For the sink experiments, we would, for example, compute the necessary and sufficient conditions for `p`'s dereference as $\texttt{p!} = \texttt{NULL} \wedge \texttt{flag!} = 0$ and `false` respectively. To illustrate the source experiment, consider the following call sites of function `f` from Figure 3:

```
void foo() {
  int* p = malloc(sizeof(int));  /*source*/
  ...
  bar(p, flag, x);
}

void bar(int* p, int flag, int x) {
  if(x > MAX) *p = -1;
  else f(p, flag);
}
```

The line marked `/*source*/` is the source of pointer `p`;the necessary condition at `p`'s source for `p` to be ultimately dereferenced is $\texttt{x} > \texttt{MAX} \vee (\texttt{x} <= \texttt{MAX} \wedge \texttt{p!} = \texttt{NULL} \wedge \texttt{flag!} = 0)$ and the sufficient condition is $\texttt{x} > \texttt{MAX}$.

The results of the sink experiments for Linux are presented in Figure 4, and the results of source experiments are given in Figure 6. The table in Figure 5 presents a summary of the results of both the source and sink experiments for OpenSSH, Samba, and Linux.

	Interprocedurally Path-sensitive			Intraprocedurally Path-sensitive		
	OpenSSH 4.3p2	**Samba 3.0.23b**	**Linux 2.6.17.1**	**OpenSSH 4.3p2**	**Samba 3.0.23b**	**Linux 2.6.17.1**
Total Reports	3	48	171	21	379	1495
Bugs	1	17	134	1	17	134
False Positives	2	25	37	20	356	1344
Undecided	0	6	17	0	6	17
Report to Bug Ratio	3	2.8	1.3	21	22.3	11.2

Figure 7. Results of null dereference experiments for the interprocedurally path-sensitive (first three columns) and intraprocedurally path-sensitive, but interprocedurally path-insensitive (last three columns) analyses

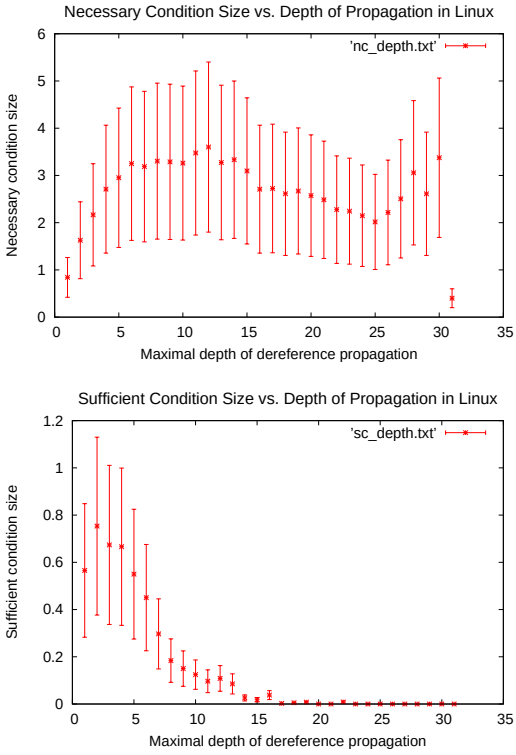

Figure 6. Necessary and sufficient condition sizes at sources vs. call chain length in Linux

The histogram in Figure 4 plots the size of necessary (resp. sufficient) conditions against the number of guards that have a necessary (resp. sufficient) condition of the given size. In this figure, red bars indicate necessary conditions, green bars indicate sufficient conditions, and note that the y-axis is drawn on a log-scale. Observe that 95% of all necessary and sufficient conditions have fewer than five subclauses, and 99% have fewer than ten subclauses, showing that necessary and sufficient conditions are small in practice. Figure 5 presents average necessary and sufficient condition sizes at sinks (rows 2 and 3) for all three applications we analyzed, confirming that average necessary and sufficient condition sizes are consistently small across all of our benchmarks. Further, the average size of necessary and sufficient conditions are considerably smaller than the average size of the original guards (which contain unobservable variables as well as the place-holder return variables representing unsolved constraints, denoted by Π in our formalism).

Figure 6 plots the maximal length of call chain from a source to any feasible sink against the size of necessary and sufficient condition sizes at sources for Linux. In this figure, the points mark average sizes, while the error bars indicate one standard deviation. First, observe that the size of necessary and sufficient conditions is small and does not grow with the length of the call chain. Second, note that the necessary condition sizes are typically larger than sufficient condition sizes; the difference is especially pronounced as the call chain length grows. Figure 5 also corroborates this trend for the other benchmark applications; average size of necessary conditions (row 4) is larger than that of sufficient conditions (row 5) at sources.

Our second experiment applies these techniques to finding null dereference errors. We chose null dereferences as an application because checking for null dereference errors with sufficient precision often requires tracking complex path conditions. To identify null dereference errors, we query the strongest necessary condition g_1 for the constraint under which a pointer p is null and the strongest necessary condition g_2 of the constraint under which p is dereferenced. A null pointer error is feasible if $SAT(g_1 \wedge g_2)$. Our implementation performs a bottom-up analysis and reports errors in the first method where a feasible path from a null value to a dereference is determined.

The first three columns of Figure 7 give the results of our fully (interprocedurally) path-sensitive null dereference experiments, and the last three columns of the same figure present the results of the intraprocedurally path-sensitive, but interprocedurally path-insensitive null dereference experiments. One important caveat is that the numbers reported here exclude error reports arising from array elements and recursive fields of data structures. Saturn does not have a sophisticated shape analysis; hence, the overwhelming majority ($> 95\%$) of errors reported for elements of unbounded data structures are false positives. However, shape analysis is an orthogonal problem; we leave incorporating shape analysis as future work. (To give the reader a rough idea of number of reports involving arrays and unbounded data structures, the number of total reports is 50 and 170 with and without full path-sensitivity respectively for OpenSSH.)

A comparison of the results of the intraprocedurally and interprocedurally path-sensitive analyses shows that our technique reduces the number of false positives by close to an order of magnitude without resorting to heuristics or compromising soundness in order to eliminate errors arising from interprocedurally correlated branches. Note that the existence of false positives for the fully path-sensitive experiments does not contradict our previous claim that our technique is complete. First, even for finite domains, our technique can only provide *relative completeness*; false positives can still arise from orthogonal sources of imprecision in the analysis (e.g., imprecise function pointer targets, inline assembly, implementation bugs, time-outs). Second, while our results are complete for finite domains, we cannot guarantee completeness for arbitrary domains. For example, when arbitrary arithmetic is involved in path constraints, our technique may fail to compute the strongest necessary and weakest sufficient conditions.

The null dereference experiments were performed on a shared cluster, making it difficult to give precise running times. A typical run with approximately 10-30 cores took around tens of minutes on SSH, a few hours on Samba, and up to more than ten hours on Linux. The running times (as well as time-out rates) of the fully path-sensitive and the intraprocedurally path-sensitive analysis were comparable for OpenSSH and Samba, but the less precise analysis took substantially longer for Linux because the fully path-sensitive analysis rules out many more interprocedurally infeasible paths, substantially reducing summary sizes.

The results of Figure 7 show that interprocedurally path-sensitive analysis is important for practical verification of software. For example, according to Figure 7, finding a single correct error report in Samba requires inspecting approximately 22.3 error reports for the interprocedurally path-insensitive analysis, while it takes 2.8 inspections to find a correct bug report with the fully path-sensitive analysis, presumably reducing user effort by a factor of 8.

9. Conclusions

We have given a method for computing the precise necessary and sufficient conditions for program properties that are fully context- and path-sensitive, including in the presence of recursive functions. We have demonstrated the practicality of our system, confirming that the approach scales to problems as computationally intensive as computing the necessary and sufficient condition for each pointer dereference in multi-million line C programs, as well as checking for null dereference errors in the largest existing open-source applications.

Acknowledgments

We would like to thank Rajeev Alur, Suhabe Bugrara, Philip Guo, Chris Unkel, and the anonymous reviewers for helpful comments on earlier drafts of this paper.

References

[1] A. Aiken, S. Bugrara, I. Dillig, T. Dillig, B. Hackett, and P. Hawkins. An overview od the SATURN project. In *Proc. Workshop on Program Analysis for Software Tools and Engineering*, pages 43–48, 2007.

[2] A. Aiken, E.L. Wimmers, and J. Palsberg. Optimal Representations of Polymorphic Types with Subtyping. *Higher-Order and Symbolic Computation*, 12(3):237–282, 1999.

[3] T. Ball and S. Rajamani. Bebop: A symbolic model checker for boolean programs. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 113–130, London, UK, 2000. Springer-Verlag.

[4] T. Ball and S. Rajamani. Automatically validating temporal safety properties of interfaces. *LNCS*, 2057:103–122, 2001.

[5] T. Ball and S. Rajamani. Bebop: a path-sensitive interprocedural dataflow engine. In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 97–103, New York, NY, USA, 2001. ACM.

[6] R. Bloem, I. Moon, K. Ravi, and F. Somenzi. Approximations for fixpoint computations in symbolic model checking.

[7] G. Boole. *An Investigation of the Laws of Thought*. Dover Publications, Incorporated, 1858.

[8] S. Bugrara and A. Aiken. Verifying the safety of user pointer dereferences. In *IEEE Symposium on Security and Privacy*, 2008.

[9] J. Burch, E. Clarke, K. McMillan, D.. Dill, and L. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proc. Symposium on Logic in Computer Science*, June 1990.

[10] D. Dill and H. Wong-Toi. Verification of real-time systems by successive over and under approximation. In *Proc. International Conference On Computer Aided Verification*, volume 939, pages 409–422, 1995.

[11] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *Proc. Conference on Programming Language Design and Implementation*, pages 57–68, 2002.

[12] I. Dillig, T. Dillig, and A. Aiken. Static error detection using semantic inconsistency inference. In *Proc. Conference on Programming Language Design and Implementation*, pages 335–345, 2007.

[13] J. Esparaza and S. Schwoon. A bdd-based model checker for recursive programs. *Lecture Notes in Computer Science*, 2102/2001:324–336, 2001.

[14] B. Hackett and A. Aiken. How is aliasing used in systems software? In *Proc. International Symposium on Foundations of Software Engineering*, pages 69–80, 2006.

[15] F. Henglein. Type inference and semi-unification. In *Proc. Conference on LISP and Functional Programming*, pages 184–197, 1988.

[16] T. Henzinger, R. Jhala, R. Majumdar, and K. McMillan. Abstractions from proofs. In *Proc. 31st Symposium on Principles of Programming Languages*, pages 232–244, 2004.

[17] F. Ivancic, Z. Yang, M.K. Ganai, A. Gupta, I. Shlyakhter, and P. Ashar. F-soft:software verification platform. *Lecture Notes in Computer Science*, 3576/2005:301–306, 2005.

[18] F. Lin. On strongest necessary and weakest sufficient conditions. In *Proc. International Conference on Principles of Knowledge Representation and Reasoning*, pages 143–159, April 2000.

[19] A. Mycroft. Polymorphic type schemes and recursive definitions. In *Proc. Colloquium on International Symposium on Programming*, pages 217–228, 1984.

[20] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61, New York, NY, USA, 1995. ACM.

[21] D. Schmidt. A calculus of logical relations for over- and underapproximating static analyses. *Science of Computer Programming*, 64(1):29–53, 2007.

[22] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. *Program Flow Analysis: Theory and Applications*, pages 189–234, 1981.

[23] Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. *SIGPLAN Not.*, 40(1):351–363, 2005.