Reasoning About the Unknown in Static Analysis

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

ABSTRACT

Static program analysis techniques cannot know certain values, such as the value of user input or network state, at analysis time. While such unknown values need to be treated as non-deterministic choices made by the program's execution environment, it is still possible to glean very useful information about how such statically unknown values may or must influence computation. We give a method for integrating such non-deterministic choices with an expressive static analysis. Interestingly, we cannot solve the resulting recursive constraints directly, but we give an exact method for answering all may and must queries. We show experimentally that the resulting solved forms are concise in practice, enabling us to apply the technique to very large programs, including an entire operating system.

1. INTRODUCTION

Preventing software errors is a central challenge in software engineering. The many tool-based approaches to the problem can be grouped roughly into two categories. *Dynamic analysis* techniques discover properties by monitoring program executions for *particular* inputs; standard testing is the most commonly used form of dynamic analysis. In contrast, a *static analysis* discovers properties that hold for *all* possible inputs; a *sound* static analysis concludes a program is error-free only if the program indeed has no errors.

Unlike dynamic analyses, sound static analyses have the advantage of never missing any potential errors, but, unfortunately, there is no free lunch: Soundness usually comes at the cost of reporting *false positives* (i.e., spurious warnings about error-free code) because static analyses must approximate some aspects of program behavior. This approximation is inevitable as analyzing even very simple properties of programs' behavior is undecidable. Hence, a key challenge for static analysis techniques is achieving a satisfactory combination of precision, soundness, and scalability by reporting as few false positives as possible while still being sound and scaling to real systems.

The original version of this paper is entitled "Sound, Complete, and Scalable Path-Sensitive Analysis" and was published in the Proceedings of Programming Language Design and Implementation (PLDI) 2008, ACM.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 2008 ACM 0001-0782/08/0X00 ...$5.00.

This goal of obtaining satisfactory precision is further complicated by the fact that certain values are simply unknown statically: For example, if a program queries the user for an input, this input appears as a non-deterministic environment choice to the static analysis. Similarly, the result of receiving arbitrary data from the network or the result of reading operating system state are all unknowns that need to be treated as non-deterministic environment choices by the analysis.

Even in the special case where all program inputs are known, static analyses still need to deal with unknowns that arise from approximating program behavior. A static analysis cannot simply carry out an exact program simulation; if nothing else, we usually want to guarantee the analysis terminates even if the program does not. Thus, static analysis always has some imprecision built in. For example, since lists, sets, and trees may have an unbounded number of elements, many static techniques do not precisely model the data structure's contents. Reading an element from a data structure is modeled as a non-deterministic choice that returns any element of the data structure. Similarly, if the chosen program abstraction cannot express non-linear arithmetic, the value of a "complicated" expression, such as `coef*a*b+size`, may also need to treated as an unknown by the static analysis.

The question of what, if any, useful information can be garnered from such unknown values is not much discussed in the literature. It is our impression that if the question is considered at all, it is left as an engineering detail in the implementation; at least, this is the approach we have taken ourselves in the past. But two observations have changed our minds: First, unknown values are astonishingly pervasive when statically analyzing programs; there are always calls to external functions not modeled by the analysis as well as approximations that lose information. Second, in our experience, analyses that do a poor job handling unknown values either end up being unscalable or too imprecise. For these reasons, we now believe a systematic approach for dealing with unknown values is a problem of the first order in the design of an expressive static analysis.

We begin by informally sketching a very simple, but imprecise, approach to dealing with unknown values in static analysis. Consider the following code snippet:

```
1: char input = get_user_input();
2: if(input == 'y') f = fopen(FILE_NAME);
3: process_file_internal(f);
4: if(input == 'y') fclose(f);
```

Suppose we want to prove that for every call to `fopen`, there is exactly one matching call to `fclose`. For the matching property to be violated, it must be the case that the value of `input` is `'y'` on line 2, but the value of `input` is not `'y'` on line 4. Since the value of the input is unknown, one simple approach is to represent the unknown value using a special abstract constant $\star$. Now, programs may have multiple sources of unknown values, all of which are represented by $\star$. Thus, $\star$ is not a particular unknown but the set of all unknowns in the program. Hence, the predicates $\star = '\texttt{y}'$ (which should be read as: $'\texttt{y}'$ is equal to some element of values represented by $\star$) and $\star \neq' \texttt{y}'$ (which should be read as: $'\texttt{y}'$ is not equal to some element of values represented by $\star$) are simultaneously satisfiable. As a result, program paths where `input` is equal to $'\texttt{y}'$ at line (2), but not equal to $'\texttt{y}'$ at line (4) (or vice versa) cannot be ruled out, and the analysis would erroneously report an error.

A more precise alternative for reasoning about unknown values is to name them using variables (called *choice variables*) that stand for a single, but unknown, value. Observe that this strategy of introducing choice variables is a refinement over the previous approach because two distinct environment choices are modeled by two distinct choice variables, β and β'. Thus, while a choice variable β may represent any value, it cannot represent two distinct values at the same time. For instance, if we introduce the choice variable β for the unknown value of the result of the call to `get_user_input` on line 1, the constraint characterizing the failure condition is $\beta = y \wedge \beta \neq y$, which is unsatisfiable, establishing that the call to `fopen` is matched by a call to `fclose`. The insight is that the use of choice variables allows the analysis to identify when two values arise from the same environment choice without imposing any restrictions on their values.

While this latter strategy allows for more precise reasoning, it leads to two difficulties –one theoretical and one practical– that the simpler, but less precise, strategy does not suffer from. Consider the following function:

```
bool query_user(bool feature_enabled) {
A:  if(!feature_enabled) return false;
B:  char input = get_user_input();
C:  if(input == 'y') return true;
D:  if(input == 'n') return false;
E:  printf("Input must be y or n!
F:    Please try again.\n");
G:  return query_user(true);
}
```

Suppose we want to know when `query_user` returns `true`. The return value of `get_user_input` is statically unknown; hence it is identified by a choice variable β. The variable `feature_enabled`, however, is definitely not a non-deterministic choice, as its value is determined by the function's caller. We represent `feature_enabled` by an *observable variable*, α, provided by callers of this function. The condition, Π, under which `query_user` returns `true` (abbreviated `T`) in any calling context, is then given by the constraint:

$$\Pi.\beta = (\alpha = \texttt{T}) \wedge (\beta = '\texttt{y}' \vee (\neg(\beta = '\texttt{n}') \wedge \Pi[\texttt{T}/\alpha] = \texttt{T})) \quad (*)$$

While this function would typically be written using a loop, the same problem arises both for loops and recursive functions, and we use a recursive function because it is easier to explain.

This formula is read as follows. The term $\alpha = \texttt{T}$ captures that the function returns `true` only if `feature_enabled` is `true` (line A). Furthermore, the user input must either be $'\texttt{y}'$ (term $\beta = '\texttt{y}'$ and line C) or it must not be $'\texttt{n}'$ (term $\neg(\beta = '\texttt{n}')$ and line D) and the recursive call on line G must return `true` (term $\Pi[\texttt{T}/\alpha]$). Observe that because the function is recursive, so is the formula. In the term $\Pi[\texttt{T}/\alpha]$, the substitution $[\texttt{T}/\alpha]$ models that on the recursive call, the formal parameter α is replaced by actual parameter `true`. Finally, the binding $\Pi.\beta$ reminds us that β is a choice variable. When the equation is unfolded to perform the substitution $[\texttt{T}/\alpha]$ we must also make the environment choice for β. The most general choice we can make is to replace β with a fresh variable β', indicating that we do not know what choice is made, but it is potentially different from any other choice on subsequent recursive calls. Thus, $\Pi[\texttt{T}/\alpha]$ unfolds to:

$$(\texttt{T} = \texttt{T}) \wedge (\beta' = '\texttt{y}' \vee (\neg(\beta' = '\texttt{n}') \wedge \Pi[\texttt{T}/\alpha]$$

While the equation (*) expresses the condition under which `query_user` returns `true`, the recursive definition means it is not immediately useful. Furthermore, it is easy to see that there is no finite non-recursive formula that is a solution of the recursive equation (*) because repeated unfolding of $\Pi[\texttt{T}/\alpha]$ introduces an infinite sequence of fresh choice variables $\beta', \beta'', \beta''', \ldots$. Hence, it is not always possible to give a finite closed-form formula describing the exact condition under which a program property holds.

On the practical side, real programs have many sources of unknowns; for example, assuming we do not reason about the internal state of the memory management system, every call to `malloc` in a C program appears as a non-deterministic choice returning either `NULL` or newly allocated memory. In practice, the number of choice variables grows rapidly with the size of the program, overwhelming the constraint solver and resulting in poor analysis scalability. Therefore, it is important to avoid tracking choice variables whenever they are unnecessary for proving a property.

Our solution to both the theoretical and the practical problems can be understood only in the larger context of why we want to perform static analysis in the first place. Choice variables allow us to create precise models of how programs interact with their environment, which is good because we never know *a priori* which parts of the program are important to analyze precisely and so introducing unnecessary imprecision anywhere in the model is potentially disastrous. But the model has more information than needed to answer most individual questions we care about; in fact, we are usually interested in only two kinds of 1-bit decision problems, *may* and *must* queries. If one is interested in proving that a program does not do something "bad" (so-called *safety properties*), then the analysis needs to ask may questions, such as "May this program dereference NULL?" or "May this program raise an exception?". On the other hand, if one is interested in proving that a program eventually does something good (so called *liveness properties*), then the analysis needs to ask must questions, such as "Must this memory be eventually freed?".

May questions can be formulated as satisfiability queries; if a formula representing the condition under which the bad event happens is satisfiable, then the program is not guaranteed to be error-free. Conversely, must questions are naturally formulated as validity queries: If a formula representing the condition under which something good happens is not

valid, then the program may violate the desired property. Hence, to answer may and must questions about programs precisely, we do not necessarily need to solve the exact formula characterizing a property, but only formulas that preserve satisfiability (for may queries) or validity (for must queries).

The key idea underlying our technique is that while choice variables add useful precision within the function invocation in which they arise, the aggregate behavior of the function can be precisely summarized in terms of only observable variables for answering may and must queries. Given a finite abstraction of the program, our technique first generates a recursive system of equations, which is precise with respect to the initial abstraction but contains choice variables. We then eliminate choice variables from this recursive system to obtain a pair of equisatisfiable and equivalid systems over only observable variables. After ensuring that satisfiability and validity are preserved under syntactic substitution, we then solve the two recursive systems via standard fixed-point computation. The final result is a *bracketing constraint* $\langle \phi_{NC}, \phi_{SC} \rangle$ for each initial equation, corresponding to closed-form strongest necessary and weakest sufficient conditions.

We demonstrate experimentally that the resulting bracketing constraints are small in practice and, most surprisingly, do not grow in the size of the program, allowing our technique to scale to analyzing programs as large as the entire Linux kernel. We also apply this technique for finding null dereference errors in large open source C applications and show that this technique is useful for reducing the number of false positives by an order of magnitude.

2. FROM PROGRAMS TO CONSTRAINTS

As mentioned in Section 1, static analyses operate on a model or abstraction of the program rather than the program itself. In this paper, we consider a family of finite abstractions where each variable has one of abstract values $C_1, \ldots, C_k$. These abstract values can be any fixed set of predicates, typestates, dataflow values, or any chosen finite domain. We consider a language with abstract values $C_1, \ldots, C_k$; while simple, this language is sufficiently expressive to illustrate the main ideas of our techniques:

$$\begin{array}{lll} \textit{Program } P & ::= & F^+ \\ \textit{Function } F & ::= & \texttt{define f(x)} = E \\ \textit{Expression } E & ::= & \texttt{true} \mid \texttt{false} \mid \texttt{C}_\texttt{i} \mid \texttt{x} \mid \texttt{f}(E) \\ & & \mid \texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3 \\ & & \mid \texttt{let x} = E_1 \texttt{ in } E_2 \\ & & \mid E_1 = E_2 \mid E_1 \wedge E_2 \mid E_1 \vee E_2 \mid \neg E \end{array}$$

Expressions are `true`, `false`, *abstract values* $\texttt{C}_\texttt{i}$, variables `x`, function calls, conditional expressions, let bindings and comparisons between two expressions. Boolean-valued expressions can be composed using the standard boolean connectives, $\wedge$, $\vee$, and $\neg$. In this language, we model unknown values 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 choice variables. Observe that this language has an expressive set of predicates used in conditionals, so the condition under which some program property holds may be non-trivial.

To be specific, in the remainder of this paper, we consider the program properties "*May* a given function return constant (i.e., abstract value) C_i?" and "*Must* a given function return constant C_i?". Hence, our goal is to compute the constraint under which each function returns constant C_i. These constraints are of the following form:

Definition 1 (*Constraints*).

$$\begin{array}{lll} \text{Equation } \mathcal{E} & ::= & [\vec{\Pi}_i].\vec{\beta} = [\vec{\mathcal{F}}_i] \\ \text{Constraint } \mathcal{F} & ::= & (s_1 = s_2) \mid \Pi[C_i/\alpha] \\ & & \mid \mathcal{F}_1 \wedge \mathcal{F}_2 \mid \mathcal{F}_1 \vee \mathcal{F}_2 \mid \neg\mathcal{F} \\ \text{Symbol } s & ::= & \alpha \mid \beta \mid C_i \end{array}$$

Symbols s in the constraint language are abstract values C_i, choice variables β whose corresponding abstract values are unknown, and observable variables α representing function inputs provided by callers. Because the values of inputs to each function f are represented by variables α, the constraints generated by the analysis are polymorphic, i.e., can be used in any calling context of f. Constraints $\mathcal{F}$ are equalities between symbols ($s_1 = s_2$), constraint variables with a substitution $\Pi[C_i/\alpha]$, or boolean combinations of constraints. The substitutions $[C_i/\alpha]$ on constraint variables are used for the substitution of formals by actuals, and recall that the vector of choice variables $\vec{\beta}$ named with the Π variable is replaced by a vector of fresh choice variables $\vec{\beta}'$ in each unfolding of the equation. More formally, if $\Pi.\vec{\beta} = \mathcal{F}$, then:

$$\Pi[C_i/\alpha] = \mathcal{F}[C_i/\alpha][\vec{\beta}'/\vec{\beta}] \quad (\vec{\beta}' \;\; \textit{fresh})$$

This renaming is necessary both to avoid naming collisions and to model that a different environment choice may be made on different recursive invocations. Constraints express the condition under which a function f with input α returns a particular abstract value C_i; we usually index the corresponding constraint variable $\Pi_{f,\alpha,C}$ for clarity. So, for example, if there are only two abstract values C_1 and C_2, the equation

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

describes the function f that always returns $\texttt{C}_1$, 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 has abstract value C_2 and vice versa. As a final example, the function

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

where the unbound variable `y` models a non-deterministic choice is described by the equation:

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

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.

Our goal is to generate constraints characterizing the condition under which a given function returns an abstract value C_i. Figure 1 presents most of the constraint inference rules for the language given above; the remaining rules are omitted for lack of space but are all straightforward analogs of the rules shown. In these inference rules, an environment A maps program variables to variables α, β in the constraint language. Rules 1-5 prove judgments $A \vdash_b e : \mathcal{F}$ where $b \in \{\textit{true}, \textit{false}\}$, 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 C_i. Finally,

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

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

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

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

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

$$(6)\quad \frac{}{A \vdash_{C_i} C_i : true}$$

$$(7)\quad \frac{i \neq j}{A \vdash_{C_i} C_j : false}$$

$$(8)\quad \frac{A(v) = \varphi \quad (\varphi \in \{\alpha, \beta\})}{A \vdash_{C_i} v : (\varphi = C_i)}$$

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

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

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

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

Figure 1: Inference Rules

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 more detail. In Rule 3, two expressions e_1 and e_2 are equal whenever both have the same abstract value. Rule 8 says that if under environment A, the abstract value of variable x is represented by constraint variable α, then x has abstract value C_i only if $\alpha = C_i$. Rule 11 presents the rule for function calls: If the input to function f has the abstract value C_k under constraint $\mathcal{F}_k$, and the constraint under which f returns C_i is Π_{f,α,C_i}, then $f(e)$ evaluates to 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:*

```
define f(x) = if ((x = C1) ∨ (y = C2)) then C1 else f(C1)
```

where `y` *models an environment choice and the only abstract values are* `C1` *and* `C2`*. Then*

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

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

We briefly sketch the semantics of constraints. Constraints are interpreted over the standard four-point lattice with $\bot \leq$ *true*, *false*, $\top$ and $\bot$, *true*, *false* $\leq \top$, where $\wedge$ is meet, $\vee$ is join, and $\neg\bot = \bot$, $\neg\top = \top$, $\neg$*true* = *false*, and $\neg$*false* = *true*. Given an assignment θ for the choice variables β, 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 $\top$) semantics. The value $\bot$ in the least fixed point semantics (resp. $\top$ in the greatest fixed point) represents non-termination of the analyzed program.

2.1 Reduction to 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. As outlined in Section 1, the algorithm has four major steps:

- eliminate choice variables by extracting strongest necessary and weakest sufficient conditions;
- rewrite the equations to preserve satisfiability/validity under substitution;
- eliminate recursion by a fixed point computation;
- finally, apply a decision procedure to the closed-form equations.

Because our abstraction is finite, constraints from Definition 1 can be encoded using boolean logic, and thus our target decision procedure for the last step is boolean SAT. We must at some point translate the constraints from Figure 1 into equivalent boolean constraints; we perform this translation first, before performing any of the steps above.

For every variable φ ($\varphi \in \{\alpha, \beta\}$) in the constraint language, we introduce boolean variables $\varphi_{i1}, ..., \varphi_{in}$ such that φ_{ij} is *true* if and only if $\varphi_i = C_j$. 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 $s_1 = s_2$ occurring in the constraints as:

$$\begin{array}{lcl} C_i = C_i & \Leftrightarrow & true \\ C_i = C_j & \Leftrightarrow & false \quad i \neq j \\ \varphi_i = C_j & \Leftrightarrow & \varphi_{ij} \end{array}$$

Note that subexpressions of the form $\varphi_i = \varphi_j$ never appear in the constraints generated by the system of Figure 1. We replace every substitution $[C_j/\alpha_i]$ by the boolean substitution $[true/\alpha_{ij}]$ and $[false/\alpha_{ik}]$ 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 $\beta = C_2$):*

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

Note that rules 3, 10, 11, and 12 implicitly quantify over multiple hypotheses; we have omitted explicit quantifiers to avoid cluttering the rules.

In the general case, the constraints from Figure 1 result in a recursive system of boolean constraints of the following form:

System of Equations 1.

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

where $\vec{\Pi} = \langle \Pi_{f_1,\alpha,C_1}, ..., \Pi_{f_k,\alpha,C_n} \rangle$ and $b_i \in \{true, false\}$ and the ϕ's are quantifier-free formulas over $\vec{\beta}$, $\vec{\alpha}$, and $\vec{\Pi}$.

Observe that any solution to the constraints generated according to the rules from Figure 1 must assign exactly one abstract value to each variable. More specifically, in the original semantics, $\varphi = C_i \land \varphi = C_j$ is unsatisfiable for any i, j such that $i \neq j$, and $\bigvee_i \varphi = C_i$ is valid; however, in the boolean encoding $\varphi_i \land \varphi_j$ and $\neg \bigvee_i \varphi_i$ are both still satisfiable. Hence, to encode these implicit uniqueness and existence axioms of the original constraints, we define satisfiability and validity in the following modified way:

$$SAT^*(\phi) \equiv SAT(\phi \land \psi_{exist} \land \psi_{unique})$$
$$VALID^*(\phi) \equiv (\{\psi_{exist}\} \cup \{\psi_{unique}\} \models \phi)$$

where ϕ_{exist} and ϕ_{unique} are defined as:

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

3. STRONGEST NECESSARY AND WEAKEST SUFFICIENT CONDITIONS

As discussed in previous sections, a key step in our algorithm is extracting necessary/sufficient conditions from a system of constraints E. The necessary (resp. sufficient) conditions should be satisfiable (resp. valid) if and only if E 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 $\mathcal{V}^+(\phi)$ to denote the set of observable variables in ϕ, and $\mathcal{V}^-(\phi)$ to denote the set of choice variables in ϕ.

Definition 2. *Let ϕ be a quantifier-free formula. We say $\lceil\phi\rceil$ is the* strongest observable necessary condition *for ϕ if:*

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

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 choice variables.

```
1. void f(int* p, int flag) {
2.   if(!p || !flag) return;
3.   char* buf = malloc(sizeof(char));
4.   if(!buf) return;
5.   *buf = getUserInput();
6.   if(*buf=='i')
7.     *p = 1;
8. }
```

Figure 2: Example code.

Definition 3. *Let ϕ be a quantifier-free formula. We say $\lfloor\phi\rfloor$ is the* weakest observable sufficient condition *for ϕ if:*

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

Let ϕ be the condition under which some program property P holds. Then, by virtue of $\lceil\phi\rceil$ being a strongest necessary condition, querying the satisfiability of $\lceil\phi\rceil$ is equivalent to querying the satisfiability of the original constraint ϕ for deciding if property P *may* hold. Since $\lceil\phi\rceil$ is a necessary condition for ϕ, the satisfiability of ϕ implies the satisfiability of $\lceil\phi\rceil$. More interestingly, because $\lceil\phi\rceil$ is the strongest such necessary condition, the satisfiability of $\lceil\phi\rceil$ also implies the satisfiability of ϕ; otherwise, a stronger necessary condition would be *false*. Analogously, querying the validity of $\lfloor\phi\rfloor$ is equivalent to querying the validity of the original constraint ϕ for deciding if property P *must* hold.

One can think of strongest necessary and weakest sufficient conditions of ϕ as defining 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 choice variables and ϕ is not equivalent to *true* or *false*, then the best possible bounds are $\lceil\phi\rceil = true$ and $\lfloor\phi\rfloor = false$. Intuitively, the "difference" between strongest necessary and weakest sufficient conditions defines the amount of unknown information present in the original formula.

We now continue with an informal example illustrating the usefulness of strongest observable necessary and weakest sufficient conditions for statically analyzing programs.

Example 3. *Consider the implementation of* `f` *given in Figure 2, and suppose we want to determine the condition under which pointer* `p` *is dereferenced in* `f`*. It is easy to see that the exact condition for* `p`*'s dereference is given by the constraint:*

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

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

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

On the other hand, the weakest observable sufficient condition for the dereference is `false`*, which makes sense because no restriction on the arguments to* `f` *can guarantee that* `p` *is dereferenced. Observe that these strongest necessary and weakest sufficient conditions are as precise as the original formula for deciding whether* `p` *is dereferenced by* `f` *at any call site of* `f`*, and furthermore, these formulas are much more concise than the original formula.*

4. SOLVING THE CONSTRAINTS

In this section, we now return to the problem of computing strongest necessary and weakest sufficient conditions containing only observable variables for each Π_{α,f_i,C_j} from System of Equations 1. Our algorithm first eliminates the choice variables from every formula. We then manipulate the system to preserve strongest necessary (weakest sufficient) conditions under substitution (Section 4.2). Finally, we solve the equations to eliminate recursive constraints (Section 4.3), 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.

4.1 Eliminating Choice Variables

To eliminate the choice variables from the formulas in Figure 1, we use the following well-known result for computing strongest necessary and weakest sufficient conditions for boolean formulas [4]:

LEMMA 1. *The strongest necessary and weakest sufficient conditions of boolean formula ϕ not containing variable β are given by:*

$$\begin{array}{rcl} \text{SNC}(\phi,\beta) & \equiv & \phi[\text{true}/\beta] \vee \phi[\text{false}/\beta] \\ \text{WSC}(\phi,\beta) & \equiv & \phi[\text{true}/\beta] \wedge \phi[\text{false}/\beta] \end{array}$$

Since our definition of satisfiability and validity must also take into account the implicit existence and uniqueness conditions, this standard way of computing strongest necessary and weakest sufficient conditions of boolean formulas must be slightly modified. In particular, let β be a choice variable to be eliminated, and let ψ_{exist} and ψ_{unique} represent the existence and uniqueness conditions involving β. Then, we compute strongest necessary and weakest sufficient conditions as follows:

$$\begin{array}{rcl} SNC^*(\phi,\beta) & \equiv & (\phi \wedge \psi_{exist} \wedge \psi_{unique})[true/\beta] \vee \\ & & (\phi \wedge \psi_{exist} \wedge \psi_{unique})[false/\beta] \\ WSC^*(\phi,\beta) & \equiv & (\phi \vee \neg\psi_{exist} \vee \neg\psi_{unique})[true/\beta] \wedge \\ & & (\phi \vee \neg\psi_{exist} \vee \neg\psi_{unique})[false/\beta] \end{array}$$

After applying these elimination procedures to the constraint system from Figure 1, we obtain two distinct sets of equations of the form:

SYSTEM OF EQUATIONS 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}.

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

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

The reader can verify that the strongest necessary condition for Π_{f,α,C_1} is true. *The existence and uniqueness constraints are omitted since they are redundant.*

4.2 Preservation Under Substitution

Our goal is to solve the recursive system given in System of Equations 2 by an iterative, fixed point computation. However, there is a problem: as it stands, System of Equations 2 may not preserve strongest necessary and weakest sufficient conditions under substitution for two reasons:

- Strongest necessary and weakest sufficient conditions are not preserved under negation (i.e., $\neg\lceil\phi\rceil \not\Leftrightarrow \lceil\neg\phi\rceil$ and $\neg\lfloor\phi\rfloor \not\Leftrightarrow \lfloor\neg\phi\rfloor$), and the formulas from System of Equations 2 contain negated return (Π) variables. Therefore, substituting $\neg\Pi$ by $\neg\lceil\Pi\rceil$ and $\neg\lfloor\Pi\rfloor$ would yield incorrect necessary and sufficient conditions, respectively.
- The formulas from System of Equations 2 may contain contradictions and tautologies involving return variables, causing the formula to be weakened (for necessary conditions) and strengthened (for sufficient conditions) as a result of substituting the return variables with their respective necessary and sufficient conditions. As a result, the obtained necessary (resp. sufficient) conditions may not be as strong (resp. as weak) as possible.

Fortunately, both of these problems can be remedied. For the first problem, observe that while $\neg\lceil\phi\rceil \not\Leftrightarrow \lceil\neg\phi\rceil$ and $\neg\lfloor\phi\rfloor \not\Leftrightarrow \lfloor\neg\phi\rfloor$, the following equivalences do hold:

$$\lceil\neg\phi\rceil \Leftrightarrow \neg\lfloor\phi\rfloor \qquad \lfloor\neg\phi\rfloor \Leftrightarrow \neg\lceil\phi\rceil$$

In other words, the strongest necessary condition of $\neg\phi$ is the negation of the weakest sufficient condition of ϕ, and similarly, the weakest sufficient condition of $\neg\phi$ is the negation of the strongest necessary condition of ϕ. Hence, by simultaneously computing strongest necessary and weakest sufficient conditions, one can solve the first problem using the above equivalences.

To overcome the second problem, an obvious solution is to convert the formula to disjunctive normal form and drop contradictions before applying a substitution in the case of strongest necessary conditions. Similarly, for weakest sufficient conditions, the formula may be converted to conjunctive normal form and tautologies can be removed. This rewrite explicitly enforces any contradictions and tautologies present in the original formula such that substituting the Π variables with their necessary (resp. sufficient) conditions cannot weaken (resp. strengthen) the solution.

4.3 Eliminating Recursion

Since we now have a way of preserving strongest necessary and weakest sufficient conditions under substitution, it is possible to obtain a closed form solution containing only observable variables to System of Equations 2 using a standard fixed point computation technique. To compute a least fixed point, we use the following lattice:

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

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. This results in a system of bracketing constraints of the form:

SYSTEM OF EQUATIONS 3.

$$\langle E_{\rm NC}, E_{\rm SC}\rangle = \begin{bmatrix} \langle \lceil \Pi_{f_1,\alpha,C_1} \rceil, \lfloor \Pi_{f_1,\alpha,C_1} \rfloor \rangle & = & \langle \phi'_{11}(\vec{\alpha_1}), \phi''_{11}(\vec{\alpha_1}) \rangle \\ & \vdots & \\ \langle \lceil \Pi_{f_k,\alpha,C_n} \rceil, \lfloor \Pi_{f_k,\alpha,C_n} \rfloor \rangle & = & \langle \phi'_{kn}(\vec{\alpha_k}), \phi''_{kn}(\vec{\alpha_k}) \rangle \end{bmatrix}$$

Recall from Section 2 that the original constraints have four possible meanings, namely $\bot$, *true*, *false*, and $\top$, while the resulting closed-form strong necessary and weakest sufficient conditions evaluate to either *true* or *false*. This means that in some cases involving non-terminating program paths, the original system of equations may have meaning $\bot$ in least fixed-point semantics (or $\top$ in greatest fixed-point semantics), but the algorithm presented in this paper may return either *true* or *false*, depending on whether a greatest or least fixed point is computed. Hence, our results are qualified by the assumption that the program terminates.

EXAMPLE 5. *Recall that in Example 4 we computed* $\lfloor \Pi_{f,\alpha,C_1} \rfloor$ *for the function* `f` *defined in Example 1 as:*

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

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.*

5. LIMITATIONS

While the technique proposed in this paper yields the strongest necessary and weakest sufficient conditions for a property P with respect to a finite abstraction, it is not precise for separately tracking the conditions for two distinct properties P_1 and P_2 and then combining the individual results. In particular, if ϕ_1 and ϕ_2 are the strongest necessary conditions for P_1 and P_2 respectively, then $\phi_1 \wedge \phi_2$ does *not* yield the strongest necessary condition for P_1 and P_2 to hold together because strongest necessary conditions do not distribute over conjunctions, and weakest sufficient conditions do not distribute over disjunctions. Hence, if one is interested in combining reasoning about two distinct properties, it is necessary to compute strongest necessary and weakest sufficient conditions for the combined property.

While it is important in our technique that the set of possible values can be exhaustively enumerated (to guarantee the convergence of the fixed point computation and to be able to convert the constraints to boolean logic), 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. However, the proposed technique is not complete for arbitrary non-finite domains.

6. EXPERIMENTAL RESULTS

We have implemented our method in Saturn, a static analysis framework designed for checking properties of C programs [1]. As mentioned in Section 1, sources of imprecision in the analysis appear as non-deterministic choices; in Saturn, sources of imprecision include, but are not limited to,

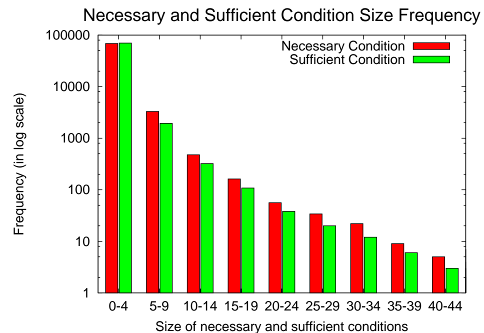

Figure 3: Frequency of necessary and sufficient condition sizes (in terms of the number of boolean connectives) at sinks for Linux

	Linux 2.6.17.1	Samba 3.0.23b	OpenSSH 4.3p2
Average 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 4: Necessary and sufficient condition sizes (in terms of number of boolean connectives in the formula) for pointer dereferences.

reads from unbounded data structures, arithmetic, imprecise function pointer targets, imprecise loop invariants, and in-line assembly; all of these sources of imprecision in the analysis are treated as choice variables.

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 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 2, consider the pointer dereference (sink) at line 7. For the sink experiments, we would, for example, compute the necessary and sufficient conditions for `p`'s dereference as `p! = NULL` $\wedge$ `flag! = 0` and `false` respectively. To illustrate the source experiment, consider the following call site of function `f` from Figure 2:

```
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 3. The table in Figure 4 presents a sum-

	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 5: Results of null dereference experiments

mary of the results of both the source and sink experiments for OpenSSH, Samba, and Linux. The histogram in Figure 3 plots the size of necessary (resp. sufficient) conditions against the number of constraints 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 4 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.

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. In the results presented in Figure 5, we compare two different set-ups: In the *interprocedurally path-sensitive* analysis, we use the technique described in the paper, computing strongest necessary conditions for a null pointer to be dereferenced. In the second setup (i.e., the *intraprocedurally path-sensitive* case), for each function, we only compute which pointers may be dereferenced in that function, but we do not track the condition under which pointers are dereferenced across functions. We believe this comparison is useful in quantifying the benefit of the technique proposed in the paper because, without the elimination of choice variables, (i) the interprocedurally path-sensitive analysis may not even terminate, and (ii) the number of choice variables grows linearly in the size of the program, overwhelming the constraint solver. In fact, for this reason, all previous analyses written in Saturn were either interprocedurally path-insensitive or adopted incomplete heuristics to decide which conditions to track across function boundaries [1].

The first three columns of Figure 5 give the results of the experiments for the first set-up, and the last three columns of the same figure present the results of the second set-up. 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 which we neither address nor evaluate in this work.

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 interprocedural dependencies. Note that the existence of false positives does not contradict our previous claim that our technique is complete. First, even for finite domains, our technique can provide only *relative completeness*; false positives can still arise from orthogonal sources of imprecision in the analysis. Second, while our results are complete for finite domains, we cannot guarantee completeness for arbitrary domains.

7. CONCLUSION

We have presented a method for systematically reasoning about unknown values in static analysis systems. We argued that, while representing unknown values by choice variables adds useful precision by correlating multiple uses of the same unknown value, eliminating these choice variables at function boundaries is necessary to avoid both scalability as well as termination problems. We have presented a technique to eliminate these choice variables with no loss of information for answering may and must queries about program properties. We have also experimentally demonstrated that analyzing unknown values in this way leads to much better precision and better scalability.

8. REFERENCES

[1] A. Aiken, S. Bugrara, I. Dillig, T. Dillig, B. Hackett, and P. Hawkins. An overview of the SATURN project. In *PASTE*, pages 43–48, 2007.

[2] T. Ball and S. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN, 113–130*, 2000.

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

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

[5] B. Cook, A. Gotsman, A. Podelski, A. Rybalchenko, and M. Vardi. Proving that programs eventually do something good. In *POPL*, pages 265–276, 2007.

[6] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *PLDI*, pages 57–68, 2002.

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

[8] T. Henzinger, R. Jhala, R. Majumdar, and K. McMillan. Abstractions from proofs. In *POPL*, pages 232–244, 2004.

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

[10] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, pages 49–61, 1995.

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