A MODULAR AND SYMBOLIC APPROACH TO STATIC PROGRAM ANALYSIS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Thomas Dillig
November 2011

Abstract

This thesis presents novel static analysis techniques for improving the quality of real-world software. The static analysis techniques we describe are immediately useful for uncovering errors in real code bases, as they are fully automatic, report few false alarms, and scale to large applications. The underlying machinery that allows us to develop these analyses is comprised of a symbolic SAT and SMT-based encoding of program states as well as modular, one function-at-a-time reasoning about the program.

More specifically, the contributions of this thesis are four-fold: The first contribution is a static inconsistency detection algorithm that uncovers inconsistent assumptions made by the programmer in a semantic way. Second, we present a novel and sound algorithm that performs an interprocedurally path-sensitive analysis that is capable of giving exact answers to may and must queries about the program with respect to a user-provided, finite abstraction. Third, we describe the first fully modular, summary-based pointer analysis that can systematically perform strong updates to abstract memory locations reachable through function arguments. Finally, this thesis describes an on-line constraint simplification algorithm that significantly improves the scalability of constraint-based program analysis techniques, such as the analyses outlined above.

Acknowledgements

First of all, I want to thank my Advisor, Alex Aiken, for all the guidance and advice he has provided to me over the years. Working with Alex has truly been a privilege, and I feel extremely grateful for all his support and mentorship. He has been a role-model for me in many ways, not just regarding how to conduct good research, but also regarding mentoring students and balancing work and family life. I can only hope that I will be able to pass some of these qualities on to my own students in the future.

I also want to thank David Dill for his support over many years, ranging from being my undergraduate advisor to reading paper drafts and writing my job recommendation letters. David's unique perspective from a model checking background has often enriched my research in many ways and I am thankful for all his help.

I would also like to thank Mooly Sagiv for all of his help and support over the past two years we have known each other. It was truly wonderful to work with Mooly during his sabbatical year at Stanford, and I hope we continue to collaborate in the future.

I also thank Martin Rinard and Tom Ball for their help and guidance over the course of my PhD, and especially during my job search process.

I thank my fellow PhD students at Stanford for their help and support over the years. In particular, I am grateful to Mayur Naik, Suhabe Bugrara, Philip Guo, Peter Hawkins, Adam Oliner, Brian Hackett, Sorav Bansal, and Yichen Xie for providing useful feedback on paper drafts and practice talks.

I also want to thank my family, and especially my dad, for always encouraging me in my pursuits, especially in attending a university in the United States for my undergraduate degree. I am also extremely grateful to my uncle Prof. Dr. Rolf Heimlinger for funding my undergraduate education.

I also thank all my friends for their support and encouragement throughout the years. I especially thank Aurelie Beaumel, Ana Gardea, and David Craig for always being there for me.

Finally, none of this work would have been possible without my loving wife, Isil, who is the most special person in the world for me.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Rough estimates currently put the number of software developers in the world at about 10 million, producing billions of new lines of code every year. As an increasing number of products and services come to rely on software, it becomes more and more critical that programs behave as expected. Unfortunately, the current "best practice", which involves standard testing, performs by all measures very poorly: A typical software project already spends over 50% of its development time in testing, yet most software shipped today has so many known bugs and security vulnerabilities that living with software errors has become an expensive and dangerous way of life.

One promising solution to this software quality crisis is *static analysis*. Unlike testing, static analysis examines the program at compile-time, without actually running it. In static analysis, *bug finding* techniques can improve software quality by uncovering problems before they appear in deployment and sound *verification* techniques can deliver guarantees about the absence of certain classes of errors on *all* inputs.

While automatic detection of bugs in software systems is a highly desirable goal, any static analysis technique that hopes to be practical needs to overcome three important challenges:

- **Low false alarms:** A useful static analysis should be precise enough not to overwhelm the user with many spurious error reports, known as *false alarms*. Since inspecting a single error report often takes substantial time and effort on the user's part, static analysis techniques that report a high number of false alarms are considered to be ineffective.

- **Automatibility:** A practical static analysis should not require extensive help from the user, for example, in the form of annotations. Since an analysis that requires many annotations from programmers is too expensive as measured by the amount of programmer time it consumes, static analysis techniques that are fully automatic have a much higher chance of being adopted in practice.

- **Scalability:** A useful static analysis technique should be able to analyze large, real-world software systems. A static analysis that does not scale beyong small hundred-line benchmarks is unlikely to be adopted for improving software quality in the real world.

This thesis describes novel and practical static analysis techniques that overcome this apparent trade-off between precision, scalability, and automatibility. More specifically, we consider precise and scalable approaches to static analysis that represent program states symbolically as logical formulas and perform a modular, one-function-at-a-time reasoning. By representing program states symbolically as SAT or SMT formulas, we can both take advantage of recent advances in constraint solving techniques and avoid the best-case exponential blow-up that arises from performing explicit case splits. By focusing on modular, summary-based analyses, we can both reuse analysis results in different calling contexts and achieve locality of reasoning by hiding irrelevant implementation details of a function from its callers.

1.1 Contributions

This thesis makes four main contributions in the area of static program analysis:

- Semantic inconsistency detection
- A new algorithm for interprocedurally path-sensitive analysis
- A modular, summary-based pointer analysis algorithm

- A constraint simplification algorithm that significantly improves the scalability of static analyses that symbolically represent program states as formulas, such as all the static analyses described in this thesis

In the rest of this section, we give a brief overview of each of these four contributions.

1.1.1 Semantic Inconsistency Detection

The first static analysis we consider in this thesis is an inconsistency detector which semantically identifies inconsistent assumptions made by the programmer, such as contradictory beliefs that a given pointer may be null as well as the belief that it is definitely non-null. Our inconsistency detection algorithm performs a form of sophisticated type inference and is both scalable and fully automatic. Unlike previous approaches to inconsistency detection, our static analysis is semantic and does not rely on syntactic clues at the source-code level. By encoding programmer assumptions semantically as logical formulas, the algorithm we describe discovers all inconsistent assumptions made by the programmer. Since inconsistent programmer assumptions tend to be highly correlated with real bugs, the resulting static analysis is able to uncover many bugs in real, multi-million line software applications without overwhelming the user with lots of false alarms.

1.1.2 A Novel Path-Sensitive Analysis Algorithm

The second contribution of this thesis a novel algorithm for fully *path-sensitive* static analysis. A static analysis is said to be path-sensitive if it differentiates between distinct execution paths of the program. While path-sensitive analyses are often much more precise than path-insensitive ones, they are also more expensive and less scalable. Many path-sensitive techniques try to improve their scalability by either using heuristics to select which predicates to track or by using counter-example guided abstraction refinement to lazily discover relevant predicates. However, heuristic-based techniques are not guaranteed to track all relevant predicates, while counterexample-guided techniques can fail to terminate.

In this thesis, we address the problem of performing an interprocedurally path-sensitive analysis in a sound, complete (in the sense of not missing any relevant path conditions), and scalable way. This technique differentiates between two classes of program variables, observable variables, whose values may be determined in calling contexts of a function, and unobservable variables, which represent either non- deterministic choices made by the program's execution environment or unknowns arising from imprecision in the static analysis. The key idea underlying our technique is that while unobservable 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, describing path- sensitive conditions for some program property. While this initial recursive system is not directly solvable, we show that the elimination of unobservable variables leads to a pair of solvable recursive systems that are as precise as the original system for answering may and must queries. This technique is the first fully path- sensitive program analysis that has successfully scaled to a program as large as the entire Linux kernel with over 6 million lines of code.

1.1.3 Modular Pointer Analysis

The third contribution is a flow- and context-sensitive pointer analysis that is fully modular. The analysis we describe analyzes the program in a strictly bottom-up fashion, computing polymorphic *summaries* for each function f and reusing this summary in every calling context of f. A modular pointer-analysis is advantageous over a whole-program pointer analysis because (i) it does not need to re-analyze a function for each of its call sites, (ii) localizes reasoning by hiding irrelevant internal implementation details, and (iii) allows the algorithm to be naturally parallelized, as any pair of functions with no caller-callee relationship can be independently analyzed.

The main insight underlying our technique is to represent the unknown points-to targets of locations using so-called *location variables* such that the sets of concrete locations represented by any two location variables are always disjoint. This representation allows the algorithm to soundly apply *strong updates* to location variables without knowing the calling context, which is crucial for the precision of a flow-sensitive pointer analysis. To enforce that the sets of concrete locations represented by a pair of location variables are disjoint, our algorithm constructs an efficient and symbolic encoding of all possible aliasing patterns on function entry and conditions points-to facts that arise in function f on aliasing relations at the call site of f. This insight allows us to construct the first fully modular pointer-analysis that can perform strong updates in a systematic way.

1.1.4 Constraint Simplification

Many program analysis techniques, such as all the analyses described above, use SAT and SMT formulas to symbolically encode program states or to represent the conditions under which a program property holds. However, since formulas are constructed incrementally, e.g., by taking the conjunction or disjunction of existing constraints, formulas become more and more redundant as the analysis progresses. Since solving constraints and other operations, such as quantifier elimination and substitution, are highly sensitive to formula size, redundancy in constraints substantially hinders the scalability of static analysis techniques.

Another contribution of this thesis is a simplified form representation of constraints that guarantees non-redundancy, and an algorithm for converting SAT and SMT formulas to their simplified form. Our experiments demonstrate that constraint simplification based on this algorithm increases the scalability of static analysis techniques by orders of magnitude.

Chapter 2

Semantic Inconsistency Inference

Much recent work in static analysis focuses on *source-sink* properties: For safety policy S, if S is violated when a value constructed at location l_1 is consumed at location l_2, then is there a feasible path from l_1 to l_2? If the answer is "yes", then the program has a bug (violates policy S). Some typical specifications are:

- Does a null value assigned to a pointer or reference reach a pointer dereference?
- Does any closed file reach a file read?
- Does a tainted input reach a security critical operation?

To be concrete, consider the following C-like code:

```
foo(...) {
   if (Q) p = NULL;   (1)
   ...
   bar(p);
}

bar(x) {
   if (R) *x;         (2)
   ...
}
```

The null value assigned at (1) reaches the dereference at (2) if predicates `Q` and `R` can both be true, resulting in a program crash. Several model checkers incorporating predicate abstraction and refinement [8, 10] and type-based systems [43] target such specifications. These systems work by searching for a path from a source to a sink violating the specification.

There is a complementary approach to these problems. Instead of trying to prove that a source can reach a sink, we can look at a set of sinks that a value x reaches and see if they express *inconsistent beliefs* about x [37]. In the example above, assume we did not have the function `foo` available, but that the function `bar` is:

```
bar(x) {
    if (x != NULL) *x;    (2)
    ...
    *x;                   (3)
    ...
}
```

Something is clearly not quite right with this function. At best `bar` is never called with a null value, in which case the test at (2) is just unnecessary and might confuse readers of the code about the actual possible values of `x`. At worst `bar` has a latent crashing bug waiting to happen, as the unprotected dereference at line (3) must cause an error if `x` is null.

Previous work on inconsistency checking is informal in nature, and it is not clear how it relates to standard semantics-based approaches to software analysis [37], but it is clear that relying only on the uses of a value for clues about program errors is something different from what source-sink systems do. The purpose of this chapter is to clarify what inconsistency checking is, how it is different from source-sink analysis, and to illustrate by example its potential in practice.

We propose that inconsistency checking is best thought of as a form of an older and better developed idea, type inference. Type inference systems already find type errors based only on the use of values; for example, in any functional language with type inference (e.g., ML or Haskell) the following code

$$\mathtt{x} + \mathtt{cons}(\mathtt{y}, \mathtt{x})$$

will be flagged as having a type error just because the two uses of `x` are type inconsistent (one as a number and the other as a list); note that the type declaration of `x` (the source) is not needed to discover this error. From this starting point we make the following contributions:

- The insight that checking consistency of uses is a type inference problem shows a fundamental difference between type inference and source-sink systems, such as most model checkers. Type inference systems find inconsistency errors in open programs, such as libraries (e.g., the second instance of function `bar` above, considered without a caller `foo`) that cannot be found by source-sink analyzers simply because no source exists.

- Casting many inconsistency checking problems as type inference problems requires non-standard types. The core issue is when the values at two usage sites x and y are considered to be "the same", so that x and y are checked for consistency. A particularly problematic case is pointers; we propose that if two pointers point to the same values under the same conditions then those two pointers are really the same pointer (see Section 2.4.1).

- For path-sensitive analyses there is a difficulty of how to construct appropriate predicates when there is no one source-sink path to use as a source of counterexamples for refinement. We present a method based on computing *correlations* between program predicates and values of interest.

- We conduct an extensive experiment, analyzing over 8 million lines of C source (including the entire Linux kernel) for null dereference errors. We have implemented both source-sink checking and inconsistency checking, and we find over 600 previously unknown null dereferences, the overwhelming majority of which are found by inconsistency checking. While there are limitations to our experiment (in particular, our implemented analyzer is unsound, which may affect the ratio of source-sink to inconsistency errors detected), based on the results it is our belief that inconsistency checking is valuable both because it works for open programs and because the discovered bugs are often local whereas understanding a source-sink path for the same bug appears daunting.

We begin our presentation with a small, paradigmatic language in which we develop our formal results (Section 2.1). We present both (intraprocedural) source-sink and inconsistency checking for this language (Section 2.2) and also extend our technique to an interprocedural analysis (Section 2.3). We then describe a null dereference analysis and necessary extensions for C programs (Section 2.4) and present our experimental results (Section 2.5).

2.1 Language and Inference System

This section describes a simple first order, call-by-value language we use for the formal development.

$$\begin{array}{lcl} \textit{Program } P & ::= & F^{+} \\ \textit{Function } F & ::= & \texttt{def}\, f(x_1, \ldots, x_n) = s \\ \textit{Statement } S & ::= & x \leftarrow^{\rho} C_i \mid x \leftarrow^{\rho} y \mid \texttt{check}^{\rho}\ b \mid \\ & & f(x_1, \ldots, x_n)^{\rho} \mid s_1 ;^{\rho} s_2 \mid \\ & & \ \texttt{if}^{\rho}\ b\ \ \texttt{then}\ s_1\ \texttt{else}\ s_2 \\ \textit{Condition } B & ::= & x = C_i \end{array}$$

The language has standard function definitions, assignments, statement sequences, and conditionals; the semantics is also standard and we omit a formal semantics for brevity. The only values in the language are nullary constructors (constants) $C_1, \ldots, C_n$. A condition $x = C_i$ is true if x has the value C_i. A statement $\texttt{check}^{\rho}\ x = C_i$ checks whether variable x is C_i. We use `check` statements to model requirements that a variable must have a certain value at a particular program point. In examples we sometimes need a no-op statement (e.g., to fill in a branch of an `if`); in such cases we write $\texttt{skip}^{\rho}$ to abbreviate the assignment $y \leftarrow^{\rho} y$. We also assume for simplicity that all variables that are not function arguments are assigned to before they are read, so we do not need to define how local variables are initialized.

The superscript ρ's on statements are *labels*. We assume all labels in a program are distinct, uniquely identifying statements. We often abuse our notation slightly by writing s^ρ to refer to the top-level label ρ of statement s.

The only sources (constructors) in this language are constants C_i and the only sinks (destructors) are the `check` statements. For example, the following program has a source-sink error: the source assigned at ρ_0 reaches the conflicting sink at ρ_4.

Example 1

```
 (x ←$^{\rho_0}$ C$_1$;$^{\rho_1}$
if$^{\rho_2}$ (y = C$_2$)
   then y ←$^{\rho_3}$ C$_3$
   else check$^{\rho_4}$ x = C$_2$);$^{\rho_5}$
if$^{\rho_6}$ (y = C$_1$)
   then skip$^{\rho_7}$
   else check$^{\rho_8}$ x = C$_1$
```

The language syntax allows us to define algorithms via structural induction, but it is also handy to be able to view a function definition as a control-flow graph. For each statement label ρ there are two *program points* ρ^- and ρ^+ representing the points immediately before and after the statement executes, respectively. Definition 1 defines the possible order of evaluation of statements within a function.

Definition 1 (Partial Order on Program Points) For a function $\texttt{def}\, f(x_1, \ldots, x_n) = s$, let $\prec_f$ be the smallest relation on program points in f satisfying for each sub-statement of s:

$$\begin{array}{rcl}
x \leftarrow^{\rho} \ldots & \Rightarrow & \rho^- \prec_f \rho^+ \\
\texttt{check}^{\rho} \ldots & \Rightarrow & \rho^- \prec_f \rho^+ \\
s_1^{\rho_1} ;^{\rho_0} s_2^{\rho_2} & \Rightarrow & \left\{ \begin{array}{l} \rho_0^- \prec_f \rho_1^- \\ \rho_1^+ \prec_f \rho_2^- \\ \rho_2^+ \prec_f \rho_0^+ \end{array} \right. \\
\texttt{if}^{\rho_0}\ b\ \texttt{then}\ s_1^{\rho_1}\ \texttt{else}\ s_2^{\rho_2} & \Rightarrow & \forall_{i=1,2} \left\{ \begin{array}{l} \rho_0^- \prec_f \rho_i^- \\ \rho_i^+ \prec_f \rho_0^+ \end{array} \right.
\end{array}$$

Let $\prec_f^*$ be the transitive closure of $\prec_f$. A *path* from ρ_0 to ρ_n is a sequence of labels $\rho_0, \ldots, \rho_n$ in f such that

(1) $\rho_i^- \prec_f^* \rho_{i+1}^-$ for $0 \leq i \leq n-1$

(2) the sequence is maximal between the endpoints: inserting any additional label after ρ_0 and before ρ_n violates (1).

A path is *complete* if it cannot be extended either by adding new labels before the first label or after the last label; a complete path is a path through the entire function body. For instance, in Example 1, there is a path ρ_0, ρ_2, ρ_4 because $\rho_0^- \prec \rho_0^+ \prec \rho_2^- \prec \rho_4^-$. This path can be extended in both directions to form a complete path $\rho_5, \rho_1, \rho_0, \rho_2, \rho_4, \rho_6, \rho_7$.

2.1.1 Guards

To allow for path-sensitivity in our static analysis, we construct *guards* that express program constraints. We use boolean satisfiability (SAT) as the underlying decision procedure for solving constraints; hence guards are represented as boolean formulas. In this section, we describe how to compute two kinds of guards:

- *statement guards* that describe the conditions under which a statement executes,

- *constructor guards* that describe the condition under which a variable x at a given program point evaluates to a constructor C_i. In addition, a constructor guard also encodes the source of the value C_i.

Constructor guards are functions of type

$$\mathtt{CG} = (\mathtt{Source} \times \mathtt{Int}) \rightarrow \mathtt{Guard}$$

The `Int` in the function signature corresponds to a constructor index, and the `Source` in function $\mathtt{def\ f(x_1, \ldots, x_n)}$ is either a label ρ of an assignment statement $z \xleftarrow{\rho} C_i$ in f or one of the function arguments $\mathtt{x_1}, \ldots, \mathtt{x_n}$. Sources used in constructor guards track the origin of every value in a function in terms of function arguments or constructor assignments within that function. We use $r, r', r_1, \ldots$ to range over sources.

Consider an assignment $x \xleftarrow{\rho} C_i$ with statement guard γ. The constructor guard g_x for x after the assignment is $g_x(\rho, i) = \gamma$, where γ is the statement guard for ρ, and $g_x(r, j) = \mathit{false}$ for all $r \neq \rho$ and $j \neq i$. Thus, the constructor guard encodes that immediately after the assignment the value of x is C_i from source ρ if γ is satisfied, and no other value/source combinations are possible.

We require that the formulas in the range of a constructor guard to be pairwise disjoint: if g is a constructor guard and $g(r, i) = \gamma_1$ and $g(r', j) = \gamma_2$, then $\gamma_1 \wedge \gamma_2 = \mathit{false}$ if $r \neq r'$ or $i \neq j$. This condition captures the idea that a value cannot simultaneously be two distinct constructors or come from two different sources. We can always enforce this condition by adding new unconstrained boolean variables to guards. For example, if there are only two constructors C_1 and C_2, then the constructor guard g with $g(r, 1) = \alpha$ and $g(r, 2) = \neg\alpha$ enforces disjointness; for more constructors we can use additional fresh variables. We write D_x for a fresh constructor guard associated with function argument x. By fresh, we mean that the formulas in the range of D_x share no variables with D_y for distinct variables x and y. Furthermore, $D_x(r, j) = \mathit{false}$ for all $r \neq x$; i.e., the only source of values in D_x is x.

Figure 2.1 gives inference rules for computing both statement guards and constructor guards resulting from executing a statement. An *environment* $\Gamma : \mathtt{Var} \rightarrow \mathtt{CG}$

$$(1) \qquad \Gamma, \gamma \vdash x \hookleftarrow^{\rho} C_i : \Gamma[x \leftarrow F]$$
$$\text{where } F = \lambda(r,j).\text{if } (r,j) = (\rho,i) \text{ then } \gamma \text{ else } \mathit{false}$$

$$(2) \qquad \Gamma, \gamma \vdash x \hookleftarrow^{\rho} y : \Gamma[x \leftarrow \lambda(r,j).\Gamma(y)(r,j) \wedge \gamma]$$

$$(3) \qquad \Gamma, \gamma \vdash \texttt{check}^{\rho}\ x = C_i : \Gamma$$

$$(4) \qquad \Gamma, \gamma \vdash f(x_1, \ldots, x_n)^{\rho} : \Gamma$$

$$(5) \qquad \frac{\begin{array}{c}\Gamma_0, \gamma \vdash s_1 : \Gamma_1 \\ \Gamma_1, \gamma \vdash s_2 : \Gamma_2\end{array}}{\Gamma_0, \gamma \vdash s_1 ;^{\rho} s_2 : \Gamma_2}$$

$$(6) \qquad \frac{\begin{array}{c}\pi = \bigvee_r \Gamma_0(x)(r,i) \\ \Gamma_0, \gamma \wedge \pi \vdash s_1 : \Gamma_1 \\ \Gamma_0, \gamma \wedge \neg\pi \vdash s_2 : \Gamma_2\end{array}}{\Gamma_0, \gamma \vdash \texttt{if}^{\rho}\ x = C_i\ \ \texttt{then}\ s_1\ \texttt{else}\ s_2 : \Gamma_1 \sqcup \Gamma_2}$$

$$(7) \qquad \frac{\lambda x_i.D_{x_i}, \mathit{true} \vdash s : \Gamma}{\vdash \texttt{def}\, f(x_1, \ldots, x_n) = s}$$

Figure 2.1: Computing guards.

is a map from program variables to constructor guards. For a statement s and initial environment Γ and statement guard γ, the system proves sentences of the form $\Gamma, \gamma \vdash s : \Gamma'$, where Γ' is the final environment after execution of s. Note that the inference system is purely structural; in any proof there is exactly one conclusion associated with statement s, which we can rewrite as:

$$\Gamma^{\rho^-}, \gamma^{\rho} \vdash s^{\rho} : \Gamma^{\rho^+}$$

In this way we can refer to the environments for the program points before and after ρ as well as the statement guard under which s^{ρ} is executed.

We briefly explain the rules in Figure 2.1. When a variable x is assigned a constructor C_i (rule (1)), x's constructor guard shows that it cannot have any value other than C_i from source ρ (guards for all other constructors and all other sources are *false*). Furthermore, x only has value C_i if the assignment executes (the guard γ on the assignment statement holds). The second form of assignment (rule (2)) says that the possible sources/values of x after the assignment are the possible sources/values of y before the assignment, but only if the assignment executes—the statement guard γ is added to the guard of every possible source/value pair.

A $\texttt{check}^{\rho}\ x = C_i$ statement (rule (3)) tests the predicate $(x = C_i)$ at run-time. These are the sinks in our language. The purpose of our analyses is to characterize when the run-time test can evaluate to *false*; this can model, for example, the implicit assertion that a pointer is non-null before it is dereferenced, or more generally that a value of a discriminated union type has the correct constructor (our choice of the term “constructor” is meant to suggest discriminated unions), or that a value is in the correct type-state before some operation is performed. As our interest is in when the test is *false* and not what happens as a result of the test, we define `check` statements to have no effect on the environment.

Function calls (rule (4)) also have no effect on the environment; because there are no visible side-effects of a function and no return value, function calls have no effect on the callee's state. Of course, this rule also gives us no information about `check` statements in the called function that may fail; thus, Figure 2.1 defines an intraprocedural analysis. We discuss extensions to interprocedural analysis in Section 2.3.

Rules (5) and (6) deal with compound statements. The rule for statement sequences (rule (5)) is standard. For an `if` statement (rule (6)) with statement guard γ, the guard π combines all the conditions under which $X = C_i$ from any source. The true branch is analyzed with statement guard $\gamma \wedge \pi$ and the false branch is analyzed with statement guard $\gamma \wedge \neg\pi$. The final result is a join $\Gamma_1 \sqcup \Gamma_2$ of the final environments of the two branches, defined as

$$(\Gamma_1 \sqcup \Gamma_2)(x)(r, i) = \Gamma_1(x)(r, i) \vee \Gamma_2(x)(r, i)$$

Finally, a function body (rule (7)) is analyzed in an environment where nothing is known about a function argument except that it evaluates to a single constructor at a given call site (recall that for each argument x, the guards in the range of the constructor guard for x are all disjoint).

Notice that statement guards and constructor guards are mutually dependent (e.g., rules (1) and (6)) and thus are computed simultaneously. The reason for this design decision is that the computation of statement guards is affected by side-effects of statements, which are in turn implicitly captured by constructor guards. Conversely, the condition under which a statement causes a particular side-effect to happen depends on the condition under which that statement executes; hence the computation of constructor guards make use of statement guards. As an illustration of guard computation, consider the example:

Example 2

```
if$^{\rho_0}$ $(\mathtt{x} = \mathtt{C}_1)$
  then (
      x $\leftarrow^{\rho_1}$ $\mathtt{C}_2;^{\rho_6}$
      if$^{\rho_2}$ $(\mathtt{x} = \mathtt{C}_2)$
          then check$^{\rho_3}$ $\mathtt{x} = \mathtt{C}_2$
          else skip$^{\rho_4}$ )
  else skip$^{\rho_5}$
```

Let α_1 and α_2 be formulas that represent the conditions under which the function argument `x` evaluates to `C`$_1$ and `C`$_2$ respectively at function entry. Since the statement guard at program point ρ_1 is α_1, the constructor guard $\Gamma(x)(\rho_1, 2)$ is also α_1 by Rule (1). By Rule (6), the statement guard at ρ_3 is $\alpha_1 \wedge \alpha_2$, which is *false* by our assumption that the formulas in the range of a constructor guard are disjoint.

As this example illustrates, the computation of statement guards directly allows the discovery of infeasible paths in a program.

Definition 2 (Feasibility) Let $\rho_0, \ldots, \rho_n$ be a path. Then the path is *feasible* if $SAT(\bigwedge_{0 \leq i \leq n} \gamma^{\rho_i})$.

Returning to Example 1, the path of the source-sink error ρ_0, ρ_2, ρ_4 is feasible for an appropriate initial environment, but the path $\rho_0, \rho_2, \rho_3, \rho_6, \rho_7$ is not feasible in any environment. The following lemma captures some simple but very useful facts about feasible paths.

Lemma 1 Assume $\Gamma, \gamma \vdash s : \Gamma'$ and let σ be any assignment that satisfies γ. Then there is a unique complete, feasible path including s such that σ satisfies all the statement guards on the path.

Proof 1 *The proof is by induction on the structure of s. The interesting case is when $s = (\texttt{if}^\rho\ x = C_i\ \ \texttt{then}\ s_1\ \texttt{else}\ s_2)$. From rule (6) of Figure 2.1, the final step of the derivation must be:*

$$\frac{\begin{array}{c}\pi = \bigvee_r \Gamma(x)(r, i) \\ \Gamma, \gamma \wedge \pi \vdash s_1 : \Gamma_1 \\ \Gamma, \gamma \wedge \neg\pi \vdash s_2 : \Gamma_2\end{array}}{\Gamma, \gamma \vdash \texttt{if}^\rho\ x = C_i\ \ \texttt{then}\ s_1\ \texttt{else}\ s_2 : \Gamma_1 \sqcup \Gamma_2}$$

Now either $\sigma(\gamma \wedge \pi)$ is true or $\sigma(\gamma \wedge \neg\pi)$ is true. Assume that $\sigma(\gamma \wedge \pi)$ is true. Then

$$\Gamma, \gamma \wedge \pi \vdash s_1 : \Gamma_1$$

satisfies the induction hypothesis with assignment σ, and so there is a unique complete *feasible path $\rho_1, \ldots, \rho_n$ for s_1 such that $\sigma(\gamma^{\rho_i})$ is true for all $1 \leq i \leq n$. Then $\rho, \rho_1, \ldots, \rho_n$ is the desired path for s. The case where $\sigma(\gamma \wedge \neg\pi)$ is true is symmetric.*

2.2 Error Detection

In this section we present techniques for identifying source-sink and inconsistency errors using the machinery developed in Section 2.1. Only intraprocedural techniques are discussed here; Section 2.3 extends the approach across function boundaries.

2.2.1 Source-Sink Errors

Source-sink errors arise when a value constructed at one program point reaches an unexpected destructor at a different program point. Most errors uncovered by model checking tools, and particularly model checkers based on counter-example driven refinement, are source-sink errors. This class of errors includes, for example, type-state properties, such as errors that arise from dereferencing a pointer that has been assigned to null, or using a tainted input in a security critical operation.

Definition 3 (Source-Sink Error) Consider the sub-derivation for a `check` statement:

$$\Gamma^{\rho^-}, \gamma^\rho \vdash \texttt{check}^\rho\ x = C_i : \Gamma^{\rho^+}$$

The check can fail because of a value from source ρ' if the statement is reachable when constructor C_j originating from ρ' is in the constructor guard of x for some $j \neq i$. More formally, a source-sink error arises if there is a label ρ' of an assignment statement $y \leftarrow^{\rho'} C_j$ such that

$$SAT(\gamma^\rho \wedge \bigvee_{j \neq i} \Gamma^{\rho^-}(x)(\rho', j))$$

The following lemma shows that there is always at least one feasible path corresponding to any source-sink error.

Lemma 2 Every source-sink error is included on at least one complete feasible path.

Proof 2 *Let $\psi^{\rho}_{\rho'} = \gamma^{\rho} \wedge \bigvee_{j \neq i} \Gamma^{\rho^-}(x)(\rho', j)$, and let σ be any assignment satisfying $\psi^{\rho}_{\rho'}$. Since the formula $\psi^{\rho}_{\rho'}$ is satisfiable there is at least one such σ. By Lemma 1, σ defines a unique, complete feasible path. By expanding the definition of $\psi^{\rho}_{\rho'}$ and using the fact that rule (1) in Figure 2.1 includes the statement guard in the constructor guard after the assignment, we can show that $\psi^{\rho}_{\rho'}$ satisfies both statement guards $\gamma^{\rho'}$ and γ^{ρ}. Thus, both the assignment statement and the* `check` *are on the path.*

Consider once more the program in Example 1. Assignment statement ρ_0 gives x a constructor guard where C_1 from source ρ_0 has guard *true* (just because x is assigned C_1 at ρ_0). The constructor guard of x is unchanged where a check is performed at ρ_4. Since the check is whether $x = C_2$, one of the tests for a source-sink error is:

$$SAT(\gamma^{\rho_4} \wedge \bigvee_{j \neq 2} \Gamma^{\rho_4^-}(x)(\rho_0, j))$$

Because γ^{ρ_4} is satisfiable and $\Gamma^{\rho_4^-}(x)(\rho_0, 1)$ is *true*, we have shown a source-sink error in the program.

Note that Definition 3 requires that the source be the label of an assignment statement—we do not consider function arguments as sources in computing source-sink errors, because we do not know what actual values a function argument may have while analyzing only the function body. Source-sink errors may arise if a function is called with certain arguments that cause the `check` statement to fail. Interprocedural analysis is required in this case to find the matching source, if any, that actually causes the sink to fail; we address interprocedural source-sink errors in Section 2.3.

2.2.2 Inconsistency Errors

In this section, we define inconsistencies and describe a technique for semantically detecting inconsistency errors.

Consider the following motivating example:

Example 3

```
def f(a) =
(x $\leftarrow^{\rho_0}$ a;$^{\rho_1}$
if$^{\rho_2}$ (x $=$ C$_1$)
    then check$^{\rho_3}$ x $=$ C$_1$
    else y $\leftarrow^{\rho_4}$ x);$^{\rho_5}$
check$^{\rho_6}$ a $=$ C$_1$
```

In this example, `a` and `x` are aliases for the same value because of the assignment at ρ_0. At ρ_3, x is asserted to have the value C_1 and this statement is protected by the conditional at ρ_2. The variable a is also asserted to be C_1 at ρ_6, but without the protecting test. Thus, if there actually is an environment in which this function can be called where $a \neq C_1$, an error is sure to be raised at ρ_6. The presence of the test at ρ_2 protecting the check at ρ_3 is evidence that some programmer believes there are such environments. Thus, without knowing anything about the rest of the program, it is likely that there is something wrong in this function because of the inconsistent assumptions about a and x.

This example illustrates that inconsistency errors can involve aliasing if multiple names for the same value are used inconsistently. Finding inconsistency errors means identifying a set of uses of the same value that should be compared. If we are to take aliasing into account, we cannot rely on uses of the same variable name or (more generally) syntactically identical program expressions to identify the set of uses—a semantic test for "sameness" is needed.

More formally, we define a congruence relation $v_1 \cong v_2$ that captures when two quantities v_1 and v_2 should be checked for consistency. The exact definition of $\cong$ varies with the programming language. For our toy language, an appropriate definition is that two variables at given program points are congruent if they have the same values under the same guards at those points.

Definition 4 (Congruence) Let v_1 and v_2 be two variables in the same function f, and let ρ_1^- and ρ_2^- be program points in f. Then $v_1^{\rho_1^-} \cong v_2^{\rho_2^-}$, meaning variable v_1 at program point ρ_1 is congruent to variable v_2 at program point ρ_2, if

$$\forall i. \bigvee_r \Gamma^{\rho_1^-}(v_1)(r,i) \equiv \bigvee_r \Gamma^{\rho_2^-}(v_2)(r,i)$$

Notice that we do not require that the sources of congruent variables be the same. Thus x and y can be congruent even if they are constructed completely independently; we return to this point shortly.

Definition 5 (Inconsistency Error) Consider two `check` statements $\texttt{check}^{\rho_0}\ x = C_i$ and $\texttt{check}^{\rho_1}\ y = C_i$. There is an *inconsistency error* between the two statements if the variables are congruent and one `check` can fail while the other cannot. Formally, there is an inconsistency if the following three conditions are satisfied:

$$\begin{array}{ll} (1) & x^{\rho_0^-} \cong y^{\rho_1^-} \\ (2) & \neg SAT(\gamma^{\rho_0} \wedge \bigvee_r \bigvee_{j \neq i} \Gamma^{\rho_0^-}(x)(r,j)) \\ (3) & SAT(\gamma^{\rho_1} \wedge \bigvee_r \bigvee_{j \neq i} (\Gamma^{\rho_1^-}(x)(r,j)) \end{array}$$

Condition (2) says that it is not the case that the statement guard at ρ_0 can hold and x has some value other than C_i. Condition (3) says that there is at least one solution where the statement guard at ρ_1 holds and y has some value other than C_i.

Returning to Example 3 above, at point ρ_6^- the variable a has constructor guard D_a (the original guards for a, as there are no assignments to a in the function) and at point ρ_3^- the variable x has the same guards because of the assignment at ρ_0. Thus $a^{\rho_6^-} \cong x^{\rho_3^-}$, satisfying condition (1). Now the statement guard at ρ_3 includes a conjunct $\Gamma^{\rho_3^-}(x)(a,1)$, which is disjoint with any guard $\Gamma^{\rho_3^-}(x)(r,j)$ for $j \neq 1$ (recall Section 2.1.1). Hence, the `check` statement at ρ_3 cannot fail, and condition (2) is satisfied. Finally, the statement guard at ρ_6^- is just *true*, and so condition (3) is also satisfied.

As noted above, our definition of congruence does not require any dataflow relationship between the two variables—variables with different sources may still be congruent. Thus, unlike in Example 3, two congruent variables may not even have a common source. At first look, this definition of congruence seems too permissive in that it allows variables that apparently coincidentally share the same values to be compared. We argue that even when two variables don't share a common source, an inconsistency still exists. Consider the following example:

Example 4

```
def f(a) =
   if(a=$C_1$)
      then x $\leftarrow$ $C_1$
      else x $\leftarrow$ $C_2$;
   if(a=$C_1$)
      then y $\leftarrow$ $C_1$
      else y $\leftarrow$ $C_2$;
   check(x=$C_2$);
   if(y=$C_2$)
      then check(y=$C_2$)
      else skip;
```

In this example, `x` and `y` have the same values under the same conditions, but not from the same sources. The conditional `check (y = ` C_2`)` indicates that some programmer believes there is some call site where `a` can be C_1; otherwise, `y` would always be C_1. But if this is the case, then `x` can also be C_1, and there is at least one execution trace where `check(x=`C_2`)` will fail. Hence, the above example should be classified as an inconsistency, showing that our definition of congruence is not more permissive than it should be.

Finally, note that while source-sink errors are characterized by a single feasible path, inconsistency errors are characterized by a feasible path (condition (3)) and the absence of any feasible path to a different program point (condition (2)). Thus, inconsistency inherently requires reasoning about the relationships among multiple paths, unlike source-sink error detection.

2.2.3 Intersection of Source-Sink and Inconsistency Errors

Our discussion so far highlights that source-sink and inconsistency error detection techniques are fundamentally different: First, detection of source-sink errors involves reasoning about a single program path, while the detection of inconsistencies can require reasoning about multiple paths. Second, source-sink error detection requires the source to be explicit in the source code, while inconsistency detection infers errors only from usage sites, i.e., sinks, and can therefore find errors even when the source comes from the environment.

Despite these differences, some errors can be seen both as source-sink and inconsistency errors; the following example is prototypical:

Example 5

```
 if$^{\rho_0}$(x = C$_1$)
   then check$^{\rho_1}$ x = C$_2$
   else skip$^{\rho_2}$
```

This example has an obvious error since the conditional `if(x` $=$ `C`$_1$) ensures that the `check` statement at program point ρ_1 fails. Despite the fact that there is no explicit source (i.e., a constructor assignment), the above example can be considered a source-sink error. Since `x` is known to be `C`$_1$ inside the true branch of the `if` statement, adding an extra assignment of the form `x` $\xleftarrow{\rho}$ `C`$_1$ in the true branch preserves program semantics and introduces a feasible path between the source `x` $\xleftarrow{\rho}$ `C`$_1$ and the sink `check`$^{\rho_1}$ `x` $=$ `C`$_2$.

On the other hand, we can also see this error as an inconsistency. Using the intuition that inconsistency detection is a generalization of type inference, we can introduce types `POSSIBLY_C1` and `NOT_C1`. Informally, the example does not type-check because the test `if`$^{\rho_1}$(`x` $=$ `C`$_2$) adds a type constraint that `x` has type `POSSIBLY_C1`, while the unprotected check statement assigns the `NOT_C1` type to `x`. More precisely, we can identify this error using Definition 5. Assuming that the language has only the constructors `C`$_1$ and `C`$_2$, adding the check statements `check`$^{\rho'}$ `x` $=$ `C`$_1$ and `check`$^{\rho''}$ `x` $=$ `C`$_2$ in the true and false branches of the `if` statement respectively preserves the semantics of the above program, yielding the semantically equivalent code:

```
if$^{\rho_0}$(x = C$_1$)
   then (
        check$^{\rho'}$ x = C$_1$;$^{\rho'''}$
        check$^{\rho_1}$ x = C$_2$ )
```

```
    else check$^{\rho''}$ x = C$_2$
```

This directly exposes the inconsistency in the program according to Definition 5, because the check statement at ρ_1 can fail while the one at ρ'' cannot.

2.3 Interprocedural Error Detection

In this section we discuss interprocedural extensions to our approach for detecting both source-sink and inconsistency errors. Before presenting our interprocedural analysis we first revisit what we mean by inconsistency errors; unlike source-sink errors, the definition of inconsistency must be reconsidered in the interprocedural case. Consider the following example:

Example 6

```
 def f(x) =
    if$^{\rho_0}$(x = C$_1$)
        then check$^{\rho_1}$ x = C$_1$
        else skip$^{\rho_2}$ ;$^{\rho_3}$
    g(x)$^{\rho_4}$

def g(y) =
    check$^{\rho_5}$ y = C$_1$
```

This program clearly has an inconsistency error: the `check` at ρ_1 is protected by a test at ρ_0, but the `check` in `g` on the same value is unprotected. Now consider the following, slightly different, example:

Example 7

```
 def f(x) =
    g(x)$^{\rho_0}$ ;$^{\rho_1}$
    check$^{\rho_2}$ x = C$_1$
```

```
def g(y) =
    if$^{\rho_0}$(y = $C_1$)
        then check$^{\rho_1}$ y = $C_1$
        else skip$^{\rho_2}$
```

This example simply interchanges the protected and unprotected `check` statements: now the `check` in the caller is unprotected while the callee guards the `check`. Extending our intraprocedural definition of inconsistency errors in the obvious way leads us to conclude that this example also has an inconsistency error, but this definition of inconsistency results in large numbers of false positives on real programs. The issue is that `g` may have other callers besides `f`. That is, while `f` may be safe in relying on $\mathtt{x} = \mathtt{C}_1$, other callers of `g` may pass arguments other than C_1. Defensive programming of this sort is very common in practice. A typical example is a library that does extensive checking of arguments, while client code may be written with the knowledge that certain values cannot arise.[1]

In summary, Example 6 should be considered an inconsistency error, while Example 7 should not. Thus, when comparing two uses of a value between a caller and a callee, we only consider pairs of uses where the callee `check` can fail. This decision implies that we do not need to track `check` statements that are guaranteed to succeed outside of their containing function; the only interprocedural information we need is knowledge of when a function can fail.

We use function summaries for interprocedural analysis: a summary is computed of the conditions under which a function f can fail, and this summary is then used at each call site of f to model f's behavior for the purpose of detecting source-sink and inconsistency errors. This approach is context-sensitive, since the summaries are applied separately at every call site. We begin by describing how function summaries are defined and used in a basic form (Sections 2.3.1 and 2.3.3) and then describe a significant improvement (Section 2.3.2).

[1]A similar problem arises with our definition of inconsistency in the presence of function macros. Since macros are used in many different contexts, they are often written with defensive checks. In our implementation, code resulting from a macro expansion is tagged in the parse tree as coming from a macro and treated as an inlined function.

2.3.1 Function Summaries

A function summary describes the preconditions on the execution of a function that, if satisfied, may lead to errors. Computing sound and very precise preconditions is easy in our framework; the disjunction of all the failure conditions for every `check` statement in a function characterizes exactly the condition under which some `check` will fail. Unfortunately, propagating such precise information interprocedurally is prohibitively expensive; the formulas grow very rapidly as conditions are propagated through a series of function calls.

We take a different approach to function summaries that is designed to scale while still expressing all the possible conditions under which a `check` in a function may fail. The price we pay is a loss of precision in the general case; one can construct examples for which our summaries greatly overestimate the precondition for failure. However, our summaries do precisely summarize the failure precondition of the vast majority of functions we have observed in practice.

A *function summary* S has the same signature as a constructor guard, a map from sources, in this case just function arguments, and constructor indices to guards. The interpretation of summaries is different, however. The idea is that if $S(a, k) = \pi$, then a call of f where formal parameter a is C_k can fail if the initial state of the call also satisfies predicate π. For example, in Example 6, $S_g(y, 1) = \mathit{false}$ and $S_g(y, i) = \mathit{true}$ for $i \neq 1$ captures that when the argument is C_i for any $i \neq 1$ function g may fail. In Example 7, $S_g(y, i) = \mathit{false}$ for all i expresses that the function can never fail.

Definition 6 (Function Summary) Consider a function f where

$$\frac{\lambda x_i.D_{x_i}, \mathit{true} \vdash s^{\rho_0} : \Gamma}{\vdash \mathtt{def}\, f(x_1, \ldots, x_n) = s^{\rho_0}}$$

Then

$$
\begin{array}{ll}
\multicolumn{2}{l}{(S_f(x_i, j) = \pi) \Rightarrow \Pi(x_i, j, \pi) \text{ where}} \\
\multicolumn{2}{l}{\Pi(x_i, j, \pi) \equiv} \\
(1) & \forall(\texttt{check}^{\rho_1}\ x = C_k) \text{ in } f \text{ where } k \neq j. \\
(2) & \qquad \text{if } SAT(\gamma^{\rho_1} \wedge \Gamma^{\bar{\rho_1}}(x)(x_i, j)) \text{ then} \\
(3) & \qquad\qquad (\gamma^{\rho_1} \wedge \Gamma^{\bar{\rho_1}}(x)(x_i, j)) \Rightarrow \pi
\end{array}
$$

In words, for each function argument x_i and constructor C_j, on line (1) we consider the set S of all statements $\texttt{check}^{\rho_1}\ x = C_k$ such that the `check` fails if $x = C_j$ (i.e., the condition $k \neq j$). On line (2) we further restrict our focus to the subset S' of statements in S where the `check` can fail because the source of constructor C_j is argument x_i. On line (3), for every `check` in this smaller set S', we are looking for a necessary condition π that holds whenever one of the checks in S' fails. As a result, π gives an over-approximation of the condition under which a `check` statement in `f` will fail if argument x_i is constructor C_j at some call site. In other words, if $SAT(\pi \wedge (x_i = C_j))$ for some call site, a `check` may fail in `f`.

It is easy to see that setting π to *true* always satisfies the conditions, so that $S_f(x_i, j) = \textit{true}$ for all x_i and C_j is always a correct, if very imprecise, function summary. If $S_f(x_i, j) = \textit{false}$ then no `check` in f can fail when $x_i = C_j$.

One simple strategy for computing function summaries is:

$$
S_f(x_i, j) = \begin{cases} \textit{false} & \text{if } \Pi(x_i, j, \textit{false}) \\ \textit{true} & \text{otherwise} \end{cases}
$$

The reader may easily confirm that this algorithm yields $S_g(y, 2) = \textit{true}$ and $S_g(y, 1) = \textit{false}$ for Example 6. In Section 2.3.2 we consider how to compute guards π other than *true* and *false*. Now consider a more involved example:

Example 8

```
def foo(a_1, a_2) =
    if^ρ0 (a_2 = C_2)
        then x ←^ρ1 a_2
    else x $\leftarrow^{\rho_2}$ a$_1$ ;$^{\rho_3}$
check$^{\rho_4}$ x $=$ C$_2$
```

Assume that the only constructors are C_1 and C_2. Applying the test given in Definition 6 to $S_{\texttt{foo}}(a_1, 1)$, we have:

- The single `check` statement satisfies line (1) of Definition 6 with $k = 2$.
- For line (2), γ^{ρ_4} is *true* and $\Gamma^{\bar{\rho_4}}(x)(a_1, 1)$ is satisfiable because of the assignment at ρ_2.
- For line (3), setting $\pi =$ *true* satisfies the implication.

2.3.2 Correlation Analysis

The summary generation strategy described in Section 2.3.1 has two principal strengths. First, it captures the common case where an error in the body of a function is triggered by the value of a single function argument. Second, if the only possibilities for π are *true* and *false*, then the size of summaries is guaranteed to be bounded by the product of the number of function arguments and the number of distinct constructors.

However, there are many realistic examples where this approach is not expressive enough, because there are times when programmers use two or more correlated arguments to a function; consider, for example, when one argument serves as a flag describing the state of another argument. The following example encodes such an idiom in our toy language:

Example 9

```
def f(a$_1$, a$_2$) $=$
    if$^{\rho_0}$(a$_2$ $=$ C$_1$)
        then check$^{\rho_1}$ a$_1$ $=$ C$_1$
        else skip$^{\rho_2}$
```

If the predicates of $\texttt{S}_{\texttt{f}}$ are limited to *true* and *false*, then the best we can do in this example is $\texttt{S}_{\texttt{f}}(\texttt{a}_1, 2) =$ *true*, which is rather coarse as `f`'s check does not unconditionally fail when $\mathtt{a_1} \neq \mathtt{C_1}$. A better summary would record that $\mathtt{S_f}(\mathtt{a_1}, 2) \equiv (\mathtt{a_2} = \mathtt{C_1})$, precisely capturing the necessary condition for failure when $\mathtt{a_1} = \mathtt{C_2}$. We perform a *correlation analysis* to discover such additional predicates:

Definition 7 (Correlation Analysis) Consider a function definition `def` $\mathtt{f}(\mathtt{x_1}, \ldots, \mathtt{x_n}) = \mathtt{s}$. Let $\phi_{k,h}$ be a formula for the expression $(\mathtt{x_k} = \mathtt{C_h})$.

$$S_f(x_i, j) = \bigwedge \{\phi_{k,h} | \Pi(x_i, j, \phi_{k,h})\}$$

In Example 9, we have $\gamma^{\rho_1} \equiv (\mathtt{a_2} = \mathtt{C_1})$, and so $\mathtt{S_f}(\mathtt{a_1}, 2) \equiv (\mathtt{a_2} = \mathtt{C_1})$ using the algorithm in Definition 7. Similarly, using the correlation analysis for computing a more precise summary for Example 8, we obtain $\mathtt{S_{foo}}(\mathtt{a_1}, 1) \equiv (\mathtt{a_2} = \mathtt{C_1})$.

It is instructive to compare our approach to interprocedural path sensitivity with source-sink error detectors. While full interprocedural path sensitivity may be intractable for large programs, model checking techniques have shown that computing path sensitivity in a demand-driven fashion can avoid tracking unnecessary predicates and allow analyses to scale [10, 8, 30]. However, such model checkers rely on having a full path from source to sink to drive the process of discovering the needed predicates, information we do not have available both in an inconsistency analysis and a compositional interprocedural source-sink analysis. Correlation analysis allows us to find relevant predicates that play a role in interprocedural communication by computing necessary conditions for errors to occur. The price we pay is that we restrict the space of predicates considered to ensure scalability; for example, in our toy language we only consider the predicates $\phi_{k,h}$.

2.3.3 Summary Application

Consider a function definition

$$\mathtt{def}\ \mathtt{f}(\mathtt{a_1}, \ldots, \mathtt{a_n}) = \mathtt{s}$$

and call site $\mathtt{f}(\mathtt{x_1}, \ldots, \mathtt{x_n})$ and a summary $\mathtt{S_f}$. We use the summary of `f` to model `f`'s behavior at the call site as follows. We define a new function $\mathtt{f_{summary}}(\mathtt{a_1}, \ldots, \mathtt{a_n}) = \mathtt{s}'$

where $\mathtt{s'} = \ldots; \mathtt{s_{ij}}; \ldots$ is a sequence of statements, one for every argument $\mathtt{a_i}$ and constructor $\mathtt{C_j}$. From Definition 7, $\mathtt{S_f(a_i, j)}$ must have the form

$$\mathtt{S_f(a_i, j)} = \phi_{\mathtt{k_1,l_1}} \wedge \ldots \wedge \phi_{\mathtt{k_m,l_m}}$$

Abusing our syntax slightly, we define $\mathtt{s_{ij}}$ to be:

```
if$^{\rho^{ij}}$($\mathtt{a_i = C_j}$)
    then check(($\mathtt{a_{k_1} \neq c_{l_1}}$) $\vee \ldots \vee$ ($\mathtt{a_{k_m} \neq c_{l_m}}$))
    else skip;
```

At the call site we simply replace the statement $\mathtt{f(x_1, \ldots, x_n)}$ by $\mathtt{s'[x_1/a_1, \ldots, x_n/a_n]}$. This approach, which inlines a "stub" function that approximates the error behavior of the original function, allows us to reuse the intraprocedural algorithms for detecting source-sink and inconsistency errors from Section 2.2 unchanged.

2.4 A Null Dereference Analysis

In this section, we apply our approach to the problem of detecting null dereference errors in C programs. We first present an encoding of the null dereference problem in our framework and then discuss extensions needed to analyze C.

To apply the techniques in Sections 2.1-2.3 to the problem of detecting unsafe null dereferences, we need only define the constructors and an appropriate congruence relation. Null dereference analysis is about understanding what pointers can be null, which in turn requires a reasonably precise model of all the possible values of all pointers in a program. Our C implementation incorporates a sound context-, flow- and partially path-sensitive points-to analysis for C [47]. Most points-to analyses compute a graph where the nodes V are the set of abstract locations and there is an edge $(v, v') \in E$ if location v may point to v'. The points-to analysis we use labels each points-to edge with a guard $(v, v')^g$, where g is a formula specifying under what conditions v points to v'. The value NULL is treated as a node in the graph, so $(v, \mathtt{NULL})^g$ means that v may be a NULL pointer whenever guard g is satisfied.

For the congruence relation, given a guarded points-to graph (V, E), we say that $v_1, v_2 \in V$ are congruent, $v_1 \cong v_2$, if

$$\forall v_3 \in V.(((v_1, v_3)^{g_1} \in E \Leftrightarrow (v_2, v_3)^{g_2} \in E\ )\ \wedge\ g_1 \equiv g_2)$$

That is, two pointers are equivalent if they are aliases of one another: they point to the same locations under the same conditions.

To model constructors, we classify all pointers as `NULL` or `NOT-NULL` (i.e., everything except `NULL`). Before each pointer dereference $*$`x` we insert a `check`:

$$\texttt{check}^{\rho}\ \texttt{x} = \texttt{NOT-NULL}$$

The `check` succeeds only if the `NULL` guard in `x`'s points-to graph is unsatisfiable at point ρ^-.

To illustrate how we detect null inconsistency errors in C, consider the following example:

Example 10

```
void foo(int* p, int* q, bool flag)
{
   P1.  flag = (p!= NULL);
   P2.  q = p;
   P3.  if (flag)
   P4.     *p = 8;
   P5.  *q = 4;
}
```

The assignment at `P2` ensures `p` and `q` have the same guarded points-to relationships; thus `p` $\cong$ `q`. The dereference of `p` at `P4` cannot fail because the statement guard (the test on `flag` at `P3`) guarantees that `p` is non-null. However, the dereference of `q` at `P5` can fail because the statement guard is just *true*. Thus, we detect a null inconsistency in `foo`.

	LOC	Total	Bugs	U	FP	% FP	IC	SS	Both	% Interproc	% Alias
OpenSSL 0.9.8b	339319	55	47	2	6	11.3%	40	6	1	38.3%	34.0%
Samba 3.0.23b	515689	68	46	3	19	29.2%	40	4	2	34.8%	17.4%
OpenSSH 4.3p2	154660	9	8	0	1	11.1%	6	2	0	37.4%	0.0%
Pine 4.64	372458	150	119	3	28	19.0%	105	10	4	42.0%	6.7%
MPlayer 1.0pre8	761708	119	89	2	28	23.9%	71	16	2	41.6%	30.3%
Sendmail 8.13.8	364569	9	8	0	1	11.1%	7	1	0	62.5%	12.5%
Linux 2.6.17.1	6275017	373	299	8	66	18.1%	249	38	12	27.8%	12.0%
Total	8783420	783	616	18	149	19.5%	518	77	21	34.1%	15.4 %

Figure 2.2: Experimental Results. Column labeled “Bugs” reports the number of bugs uncovered, “U” corresponds to error reports that were unresolved, the abbreviation “FP” stands for false positives, the abbreviation “IC” stands for inconsistent and the abbreviation “‘SS” stands for source-sink.

2.4.1 Extensions for C

There are features in C that are not in the toy language we have used to present our techniques. We briefly discuss the most significant extensions that are required to support analysis of C programs.

The biggest technical difference between the toy language and C is that C functions can have externally visible side-effects. In particular, for a null dereference analysis, it is necessary to estimate the set of function side-effects making locations either null or not null. We address this problem by using a separate side-effect analysis to compute sources of null (both in the return value and as a result of function side-effects) as well as to track modifications to function arguments. However, this side-effect analysis is best effort and unsound; it tracks side-effects that must result in a location being assigned null, but it does not capture all assignments that just might result in a location being assigned null. In our opinion, this is the major source of unsoundness in our implementation.

The difficulty in estimating function side-effect information lies in resolving the tension between two competing goals. First, the quantity of side-effect information is potentially enormous; computing even simple use/mod information for every function (i.e., just the set of abstract locations the function reads or writes) in a large program is intractable if the result is represented naively, because the set of side-effects of a function includes all the side-effects of functions it can call either directly or indirectly. Thus, it is necessary to aggressively summarize interprocedural side-effect information to avoid consuming space quadratic (or worse) in the size of the program. Second, the resulting information must be precise enough to yield useful results, because even small imprecisions can lead to overwhelming numbers of false positives. We are not aware of any general results on efficiently computing interprocedural side-effect information; the problem appears to be unsolved. Previous null dereference analyzers have focused on intraprocedural checking (see Section 2.6).

Another separate issue is what predicates are used by the correlation analysis to compute function summaries. In Definition 7, we considered only predicates $\phi_{k,h}$ corresponding to conditions of the form $(\mathtt{x_k} = \mathtt{C_h})$. Unfortunately, in a real programming language, there are arbitrarily many predicates of this form. For example, if a function argument `x` is an integer, it is obvious that we cannot test $\mathtt{x} = \mathtt{c}$ for every possible integer constant. Our approach is to consider only the predicates that occur inside `if` statements in the computation of π.

An orthogonal issue is the modeling of loops and recursive functions. The system defined in Sections 2.1-2.3 can be used to analyze recursive functions in a sound manner by a standard iterative fixed point computation. In our implementation for C we analyze each function only once and do not attempt to compute fixed points, in part to limit the growth in interprocedural side-effect information.[2] We have observed that the function summary guards inferred by correlation analysis are almost always very simple; in fact, conjunctions of more than two simple atomic predicates are exceedingly rare, if in fact they ever occur (we have yet to notice one with more than two clauses). Thus, we believe that very simple restrictions on the size and form of function summary guards (along with conservative approximation if those limits are exceeded) would be sufficient to ensure that a fixed point computation terminates with useful (i.e., sufficiently precise) results.

Finally, as discussed above, our system builds upon a may-alias analysis for C.

[2]Cycles of mutually recursive functions are analyzed once in an arbitrary order.

This underlying analysis is sound assuming the C program is memory safe (a standard assumption in may-alias analysis), a condition that is not checked by the alias analysis or our system.

2.5 Results

We have run our null dereference analysis on seven widely used open source projects and identified 616 null dereference issues with 149 false positive reports (an overall 19.5% false positive rate). These projects receive regular source code checking from multiple commercial bug-finding tools, and so we sought to learn whether these bugs had been previously reported. Developers for the `Samba` project confirmed that none of the `Samba` bugs had been previously found. For the other projects we did not receive such an explicit acknowledgment that the bugs were new; however, we judge from the fact that fixes were released quickly for many of the bugs shortly after our reports were filed that at least the majority of the bugs we found were previously unknown. The large majority of these bugs, 518, were found by our inconsistency analysis.

We ran our null dereference analysis on a compute cluster. Analyzing the Linux kernel with over 6 MLOC required about 4 hours using 30 CPU's, which was by far the longest time required for any of the projects. The smallest project we analyzed was OpenSSH, which took 2 minutes and 33 seconds to analyze on the same cluster. Our system makes many calls to a boolean SAT solver to test the satisfiability of the various predicates used in our analyses, and for Linux the number of SAT queries numbers in the millions. We impose a 60 second time limit for analyzing any individual function; if the analysis of a function times out, its function summary is incomplete.

Figure 2.2 summarizes our experimental results. The first column gives the number of lines of code for each project, the second column presents the total number of reports, which is classified in the following three columns into correct reports, false positives, and undecided reports (reports that we could not classify as either correct reports or as false positives, because the interpretation of these reports required a more global understanding of the code base than we had). The sixth column gives the false positive rate, which is calculated without including the undecided reports. The second group of three columns breaks down the correct reports by kind: the count of inconsistency errors excluding those also found by source-sink detection, the number of source-sink errors found also as inconsistencies, and the number of errors identified by both. The last group of two columns show the percentages of correct reports that were interprocedural and that involved pointer aliasing, respectively. Many current bug finders ignore pointer aliasing and interprocedural analysis; at least for null dereference analysis, our results show that both features are important.

We used the following methodology in classifying the error reports. First, source-sink errors resulting from dereferences of return values of functions which can potentially return null were counted once per function, not once per call site. Return values of `malloc` wrappers that can return null are often used unsafely at many call sites, resulting in a misleadingly large number of correct reports if each such call site is counted as a bug. Second, we classified inconsistency reports as correct reports if there was actually an inconsistency, not if we could prove that the inconsistency would lead to a run-time crash. Lacking a detailed global understanding of these large projects, we could often not differentiate between redundant null checks and potential crashing bugs. In our correspondence with project developers, we were told that some of the inconsistency errors are due to redundant null checks. However, a large majority of developers deemed every inconsistency, including those believed to be redundant null checks, worth fixing. The majority view was that inconsistency errors represented misunderstandings of the inconsistent function's interface and should be fixed. A large number of error reports we classified as correct were confirmed by the developers; however not all project developers gave us feedback about the validity of error reports. In such cases, the numbers in Figure 2.2 represent our best effort to classify these errors.

Figure 2.2 shows that the large majority (87.5%) of the errors are inconsistency errors (including conditional misuse errors). Since most of these inconsistency errors were immediately fixed by developers, it is our belief that semantic inconsistency detection is able to identify real errors and important interface violations in real code. Figure 2.2 also reveals that roughly a third of the overall correct reports involve interprocedural dependencies, sometimes involving many function calls, especially in the case of source-sink errors. Our initial experiments with the tool also highlight the importance of selective path-sensitivity: A first version of the analysis without path-sensitivity resulted in a high false positive rate, while experiments with full path-sensitivity had unacceptably high time-out rates. However, using the correlation analysis, the time-out rate in our experiments stayed between 0.71% and 6.4% of all functions with an acceptable false positive rate.

Another interesting observation from Figure 2.2 is that a non-negligible number of errors (roughly one-third in OpenSSL and MPlayer) involve pointer aliasing. Pointer aliasing contributes to a significant source of null pointer errors, especially inconsistency errors, in two common programming patterns. The first pattern we observed is that generic `void*` pointers are often aliased by typed pointers and aliases with different types are used with inconsistent null pointer assumptions. The other pattern is that array elements are often assigned to "convenience" pointers, which denote current, head, or tail elements of a data structure. Programmers sometimes make different null pointer assumptions when they alternate, for example, between using `array[0]` and `hd`.

The main source of false positives is imprecision in the pointer analysis we used, which collapses aggregate structures (e.g., arrays, lists) to a single abstract location. If a null pointer is assigned to any element of an aggregate data structure, it contaminates other elements of the same data structure, causing the analysis to raise false alarms whenever an element of such a contaminated data structure is dereferenced. Other contributing factors to false positives are some unmodeled constructs, such as inline assembly.

We conclude this section by presenting two sample errors reported by the analysis, which we believe to be representative of many of the error reports generated by the tool:

```
/* Linux, net/sctp/output.c, line 270 */

236 pmtu = ((packet->transport->asoc) ?
237 (packet->transport->asoc->pathmtu) :
238 (packet->transport->pathmtu));
...
269 if (sctp_chunk_is_data(chunk)) {
270   retval = sctp_packet_append_data(packet, chunk);
...
286 }

538 sctp_xmit_t sctp_packet_append_data
      (struct sctp_packet *packet,...)
540 {
...
543 struct sctp_transport *transport = packet->transport;
...
545 struct sctp_association *asoc = transport->asoc;
...
562 rwnd = asoc->peer.rwnd;
```

This example illustrates an interprocedural inconsistency error involving pointer aliasing, which might potentially cause a null dereference at line 562. On line 236, the pointer `packet->`
`transport->asoc` is compared against null and `packet` is later passed to a function which first aliases `packet->transport` as `transport` and then aliases `transport->asoc` as `asoc`, which is finally dereferenced at line 562. Despite these aliasing relationships, the caller function assumes that `packet->transport->asoc` may be null, while the called function dereferences the same pointer without ensuring it is non-null, causing the analysis to generate an inconsistency warning.

The next error illustrates an inconsistency error involving two mutually exclusive paths:

```
/* OpenSSL, e_chil.c line 1040 */
static int hwcrhk_rsa_mod_exp(BIGNUM *r, const BIGNUM *I,
                              RSA *rsa, BN_CTX *ctx)
967 {
985  if ((hptr = RSA_get_ex_data(rsa, hndidx_rsa))!= NULL)
987  {
990    if(!rsa->n){
994      goto err;
995    }
997    /* Prepare the params */
998    bn_expand2(r, rsa->n->top); /* Check for error !! */
         ...
1027  }
1028  else
1029  {
        ...
1039   /* Prepare the params */
1040   bn_expand2(r, rsa->n->top); /* Check for error !! */
       ...
1080 }
```

In the true branch of the `if` statement, the pointer `rsa->n` is checked for being null and subsequently dereferenced at line 998. On the other hand, the same pointer is dereferenced without a null check in the false branch of the same `if` statement at line 1040. The important point about this example is that detecting inconsistencies requires reasoning about multiple paths simultaneously.

2.6 Related Work

The various program analysis traditions appear to have equivalent power; for example, there is an equivalence between type systems and model checking [66]. However, these results are for closed programs. We observe that for open programs techniques that search only for a single source-sink path cannot express inconsistency errors requiring simultaneous reasoning about multiple distinct paths. We view semantic inconsistency checking as complementary to source-sink error detection; inconsistency checking can find bugs where there are multiple sinks but no sources, while source-sink checking can detect bugs between a single source and a single sink.

Our choice of the terms *constructor* and *destructor* is inspired by work on detecting uncaught exceptions in functional programs [81, 69] and soft typing [22, 2]. A core issue in both bodies of work is tracking which datatype constructors a program value may actually have at run-time. Null dereference analysis is a special case where there are only two constructors `NULL` and `NON-NULL`; our techniques could be adapted to give very precise analysis for these other applications as well.

FindBugs [50] is a widely used tool for Java that has paid particular attention to finding null dereference errors [51]. FindBugs pattern-matches on constructs that are common sources of certain error classes and performs some data-flow computation. As our implementation is for C, it is not possible to do a direct comparison. Nevertheless, it is clear that FindBugs would not find the many path-sensitive, interprocedural, and alias-dependent bugs our more semantic analyses uncover. One can also interpret our results as indicating that, at least for tools requiring no user annotations, one must move to computationally intensive models (incorporating at least path sensitivity) to do significantly better than tools like FindBugs without unusably high false positive rates.

Some approaches attack null dereferences using user annotations on function parameters and local checking of each function body. LCLint [39] uses an unsound procedure to check the safety of dereferences of parameters annotated as may-be-null. More recent annotation-based systems are much closer to being sound [42, 41]. Current annotation languages, which mark a single parameter as possibly null or definitely not null, are not expressive enough to capture the more complex path-sensitive and interprocedural relationships we observed in our experiments.

Another approach, exemplified by CCured [67], is to use a relatively inexpensive static analysis to verify the safety of many pointer dereferences statically and then to introduce dynamic checks to enforce the remaining dereferences at run-time. The unification-based type inference used in CCured would not find most of the bugs our tool detected, and while the program would at least fail in a well-defined way if the null dereference was triggered at run-time, it would still fail.

Engler et al. were the first to explicitly propose a method for finding null dereference errors based on inconsistency checking [37]. They argue that inconsistencies suggest programmer confusion and the presence of bugs, and they give some techniques for discovering inconsistencies. We observe that their notion of inconsistency is essentially the same as the idea underlying type inference systems, where inconsistent type constraints from multiple uses of a value result in a type error. Our inconsistency analysis adopts this more semantic point of view and we give purely semantic conditions for inconsistency checking, which allows our system to uncover subtler bugs involving, e.g., pointer aliasing.

Our approach to selective inter-procedural path-sensitivity is reminiscent of some selectively path-sensitive model-checking techniques. ESP, for example, only accurately tracks branches that affect relevant properties within that branch [30]. Unlike ESP, our approach is fully path-sensitive intraprocedurally, and more importantly, our analysis infers correlated predicates by computing implication relations between predicates and guards of relevant events. Model checking tools based on predicate abstraction and refinement [8, 10, 54] also achieve selective path-sensitivity by discovering relevant predicates. Such tools start with a coarse abstraction which is refined by tracking additional relevant predicates until a path is shown to be feasible or infeasible or until no new useful predicates can be discovered. As discussed in Section 2.3, our approach differs because inconsistency analysis does not have a source-sink path to use as a source of counterexamples.

Chapter 3

A Novel Path-sensitive Analysis

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.

In this chapter, we give a new approach that addresses both of these theoretical as well as practical issues. One important insight underlying our approach is that certain values in a program are simply unknown at static analysis time. 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 $\star =' \mathtt{y}'$ (which should be read as: $'\mathtt{y}'$ 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:[1]

```
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

[1]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.

`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:

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

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.

3.1 From Programs to Constraints

As mentioned earlier, static analyses operate on a model or abstraction of the program rather than the program itself. In this chapter, 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:

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

Expressions are `true`, `false`, *abstract values* `Ci`, 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 chapter, 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 8 (*Constraints)*

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

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

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

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 has abstract value C_2 and vice versa. As a final example, the function

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

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 3.1 presents most of the constraint inference rules for the language given above; the remaining rules are omitted 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 $b \in \{\mathit{true}, \mathit{false}\}$, 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, rule 12 constructs systems of equations, giving the (possibly) mutually recursive conditions under which a function returns each abstract

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

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

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

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

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

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

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

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

(9) $$\frac{\begin{array}{l} A \vdash_{\mathit{true}} e_1 : \mathcal{F}_1 \\ A \vdash_{C_i} e_2 : \mathcal{F}_2 \\ A \vdash_{C_i} e_3 : \mathcal{F}_3 \end{array}}{A \vdash_{C_i} \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 : (\mathcal{F}_1 \wedge \mathcal{F}_2) \vee (\neg\mathcal{F}_1 \wedge \mathcal{F}_3)}$$

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

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

Figure 3.1: Inference Rules

value[2].

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 11 Suppose we analyze the following function:

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

where `y` models an environment choice and the only abstract values are `C`$_1$ and `C`$_2$. Then

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

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

3.1.1 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 8. As outlined in the beginning, 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 8 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 3.1 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:

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

Note that subexpressions of the form $\varphi_i = \varphi_j$ never appear in the constraints generated by the system of Figure 3.1. 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 12 The first row of Example 11 results in the following boolean constraints (here boolean variable α_1 represents the equation $\alpha = C_1$ and β_2 represents $\beta = C_2$):

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

In the general case, the constraints from Figure 3.1 result in a recursive system of boolean constraints of the following form:

Equation 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 $b_i \in \{\mathit{true}, \mathit{false}\}$ 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 3.1 must assign exactly one abstract value to each variable. More specifically, in the original semantics, $\varphi = C_i \wedge \varphi = C_j$ is unsatisfiable for any i, j such that $i \neq j$, and $\bigvee_i \varphi = C_i$ is valid; however, in the boolean encoding $\varphi_i \wedge \varphi_j$ 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:

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

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

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

3.2 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 9 Let ϕ be a quantifier-free formula. We say $\lceil\phi\rceil$ is the *strongest observable necessary condition* for ϕ if:

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

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.

Definition 10 Let ϕ be a quantifier-free formula. We say $\lfloor\phi\rfloor$ is the *weakest observable sufficient condition* for ϕ if:

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

```
void f(int* p, int flag) {
 if(!p || !flag) return;
 char* buf = malloc(sizeof(char));
 if(!buf) return;
 *buf = getUserInput();
 if(*buf=='i')
   *p = 1;
}
```

Figure 3.2: Example code.

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 unobservable variables and ϕ is not equivalent to *true* or *false*, 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 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 13 Consider the implementation of `f` given in Figure 3.3, 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:

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

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.

3.3 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 11 Let ϕ be a quantifier-free formula. We say $\lceil\phi\rceil$ is the *strongest observable necessary condition* for ϕ if:

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

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.

Definition 12 Let ϕ be a quantifier-free formula. We say $\lfloor\phi\rfloor$ is the *weakest observable sufficient condition* for ϕ if:

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

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 unobservable 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

```
void f(int* p, int flag) {
 if(!p || !flag) return;
 char* buf = malloc(sizeof(char));
 if(!buf) return;
 *buf = getUserInput();
 if(*buf=='i')
   *p = 1;
}
```

Figure 3.3: Example code.

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 14 Consider the implementation of `f` given in Figure 3.3, 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:

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

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

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

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.

3.4 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 3.4.2). Finally, we solve the equations to eliminate recursive constraints (Section 3.4.3), 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.

3.4.1 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 [13]:

Lemma 3 The strongest necessary and weakest sufficient conditions of boolean formula ϕ not containing variable β are given by:

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

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:

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

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

Equation 2

$$E_{\mathrm{NC}} = \left[\begin{array}{rcl} \lceil \Pi_{f_1,\alpha,C_1} \rceil & = & \phi'_{11}(\vec{\alpha_1}, \lceil \vec{\Pi} \rceil [\vec{b_1}/\vec{\alpha}]) \\ & \vdots & \\ \lceil \Pi_{f_k,\alpha,C_n} \rceil & = & \phi'_{kn}(\vec{\alpha}_k, \lceil \vec{\Pi} \rceil [\vec{b_k}/\vec{\alpha}]) \end{array} \right]$$

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

Example 15 Consider the function given in Example 11, for which boolean constraints are given in Example 12. We compute the weakest sufficient condition for Π_{f,α,C_1}:

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

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.

3.4.2 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.

3.4.3 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:

Equation 3

$$\langle E_{\text{NC}}, E_{\text{SC}} \rangle = \left[\begin{array}{ccc} \langle \lceil \Pi_{f_1,\alpha,C_1} \rceil, \lfloor \Pi_{f_1,\alpha,C_1} \rfloor \rangle & = & \langle \phi'_{11}(\vec{\alpha_1}), \phi''_{11}(\vec{\alpha_1}) \rangle \\ & \vdots & \\ \langle \lceil \Pi_{f_k,\alpha,C_n} \rceil, \lfloor \Pi_{f_k,\alpha,C_n} \rfloor \rangle & = & \langle \phi'_{kn}(\vec{\alpha_k}), \phi''_{kn}(\vec{\alpha_k}) \rangle \end{array} \right]$$

Recall from Section 3.1 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 chapter 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 16 Recall that in Example 15 we computed $\lfloor \Pi_{f,\alpha,C_1} \rfloor$ for the function `f` defined in Example 11 as:

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

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.

3.5 Limitations

While the technique proposed in this chapter 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.

3.6 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.[3] For example, given an intraprocedural constraint computed by Saturn, such as $\texttt{x} = \texttt{1} \wedge \texttt{queryUser}(\texttt{y}) = \texttt{true}$ for the `queryUser` function discussed earlier, our analysis yields the interprocedural constraints $\texttt{x} = \texttt{1} \wedge \texttt{y} = \texttt{true}$ 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 3.4.2), 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 $\texttt{if}(\texttt{foo}(\texttt{a}) == \texttt{3})$ is used at some call site of `foo`, then we compute strongest necessary and weakest sufficient conditions for $\Pi_{\texttt{foo},\texttt{a},\texttt{3}}$ 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 3.4.3 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,

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

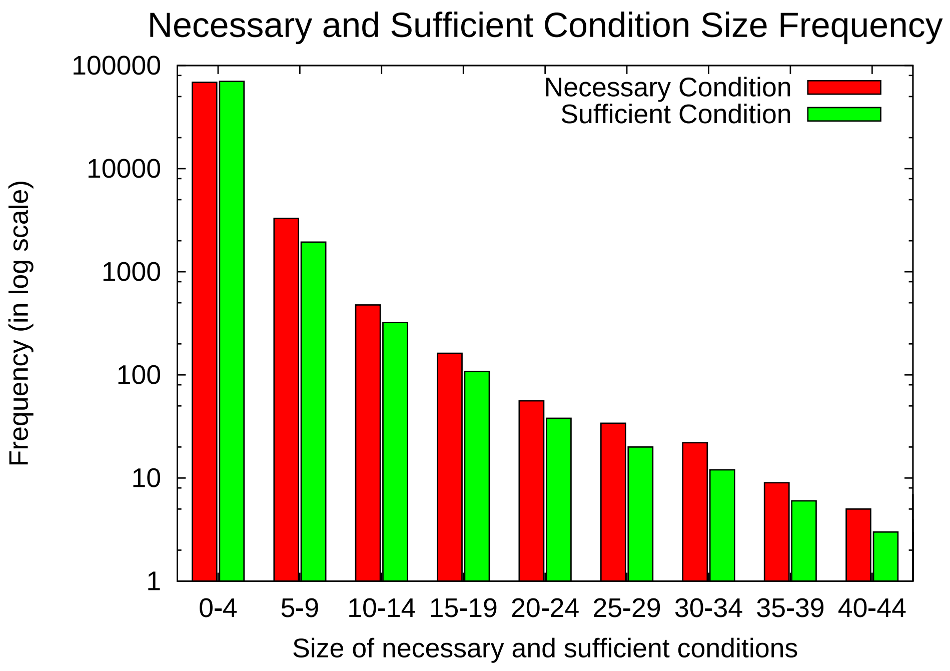


Figure 3.4: Frequency of necessary and sufficient condition sizes (in terms of the number of boolean connectives) at sinks for Linux

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:

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

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*.

	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

Figure 3.5: Necessary and sufficient condition sizes (in terms of number of boolean connectives in the formula) for pointer dereferences.

3.7 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 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 3.3, 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 $\texttt{p}! = \texttt{NULL} \wedge \texttt{flag}! = \texttt{0}$ and `false` respectively. To illustrate the source experiment, consider the following call sites of function `f` from Figure 3.3:

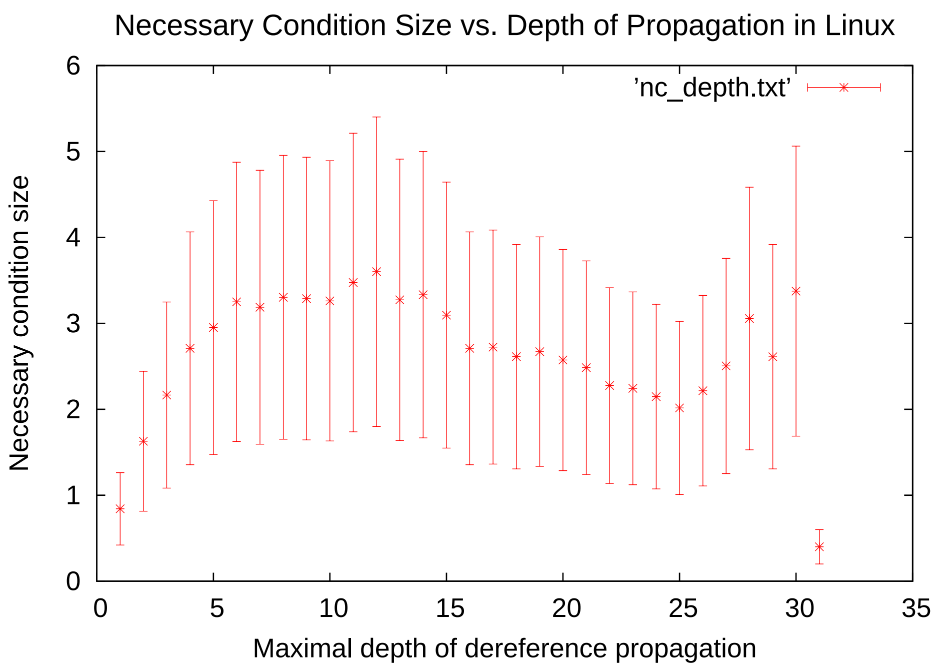
```
void foo() {
  int* p = malloc(sizeof(int));  /*source*/
  ...
  bar(p, flag, x);
}
```

```
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 $\texttt{x} > \texttt{MAX} \vee (\texttt{x} <= \texttt{MAX} \wedge \texttt{p}! = \texttt{NULL} \wedge \texttt{flag}! = \texttt{0})$ and the sufficient condition is $\texttt{x} > \texttt{MAX}$.

The results of the sink experiments for Linux are presented in Figure 3.4, and the results of source experiments are given in Figure 3.6. The table in Figure 3.5 presents a summary of the results of both the source and sink experiments for OpenSSH, Samba, and Linux. The histogram in Figure 3.4 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 3.5 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 choice variables as well as the place-holder return variables representing unsolved constraints, denoted by Π in our formalism).

Figure 3.6 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 3.5 also corroborates



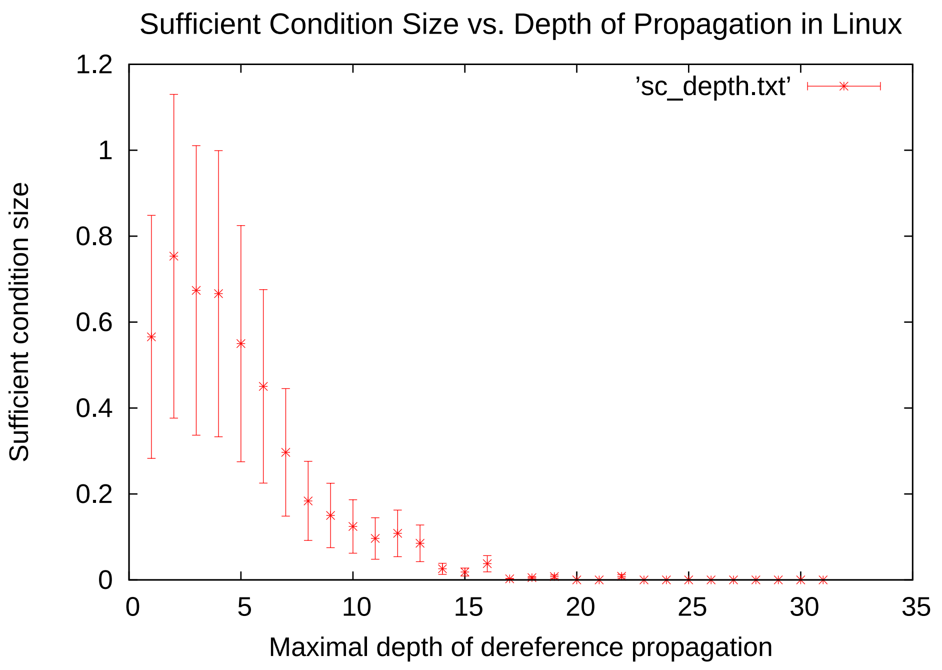


Figure 3.6: Necessary and sufficient condition sizes at sources vs. call chain length in Linux

	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 3.7: 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

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 3.7 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 3.7 show that interprocedurally path-sensitive analysis is important for practical verification of software. For example, according to Figure 3.7, 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.

3.8 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 [19].

More recent software model-checking techniques address sound and complete path- and context-sensitive analysis [7, 6, 38]. Building on techniques proposed for context-sensitivity [71, 77], Ball et al. propose Bebop, a whole-program model checking tool for boolean programs [7, 6]. 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 choice 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 eliminates these choice variables, 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 [7, 6, 38]. Another advantage of this approach is that we can use choice variables to model fixed, but unknown, parts of the environment. 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* [8, 10]. 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 [10] 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 [30]. 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 [52]. 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 [80, 33, 17, 48]. We implement the ideas proposed in this chapter 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 [76] 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 [11, 29]. 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 [13]. 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 [59].

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* [49]. Our formalization is closer to Mycroft's original work on polymorphic recursion, which represents instantiations directly as substitutions [65].

Chapter 4

Modular Heap Analysis

It is well-known that precise static reasoning about the heap is a key requirement for successful verification of real-world software. In standard imperative languages, such as Java, C, and C++, much of the interesting computation happens as values flow in and out of the heap, making it crucial to use a precise, context- and flow-sensitive heap analysis in program verification tools. Flow-sensitivity, in particular, enables *strong updates*. Informally, when analyzing an assignment `a := b`, a strong update replaces the analysis information for `a` with the analysis information for `b`. This natural rule is unsound if `a` is a *summary location*, meaning `a` may represent more than one concrete location. In previous work there is an apparent tension between scalability and precision in heap analysis:

- For scalability, it is desirable to analyze the program in pieces, for example, one function at a time. Many of the most scalable analyses in the literature are modular [1, 20].
- For precision, a large body of empirical evidence shows it is necessary to perform strong updates wherever possible [72, 32].

It is not obvious, however, how to perform strong updates in a modular heap analysis. Consider a function $\mathtt{h}(\mathtt{x},\mathtt{y})\{\mathtt{e}\}$. When analyzing `h` in isolation, we do not know how many, or which, locations `x` and `y` may point to at a call site of `h`—it may be many (if either `x` or `y` is a summary location), two, or even one (if `x` and `y` are aliases). Without this information, we cannot safely apply strong updates to `x` and `y` in `e`. Thus, while there is a large body of existing work on flow- and context-sensitive heap analysis, most algorithms for this purpose either perform a whole-program analysis or perform strong updates under very restrictive conditions.

In this chapter, we present a modular, strictly bottom-up, flow- and context-sensitive heap analysis that uses summaries to apply strong updates to heap locations at call sites. As corraborated by our experiments, strong updates are crucial for the level of precision required for successful verification. Furthermore, we are interested in a modular, summary-based analysis because it offers the following key advantages over a whole program analysis:

- *Reuse of analysis results:* A major problem with whole-program analysis is that results for a particular program component cannot be reused, since functions are analyzed in a particular context. For instance, adding a single caller to a library may require complete re-analysis of the entire library. In contrast, modular analyses allow complete reuse of analysis results because procedure summaries are valid in any context.

- *Analysis scalability:* Function summaries express a function's behavior in terms of its input/output interface, abstracting away its internal details. We show experimentally that our function summaries do not grow with program size; thus, an implementation strategy that analyzes a single function at a time, requiring only one function and its callee's summaries to be in memory, should scale to arbitrarily large programs.

- *Parallelizability:* In modular analysis, any two functions that do not have a caller/callee relationship can be analyzed in parallel. Thus, such analyses naturally exploit multi-core machines.

To illustrate our approach, consider the following simple function `f` along with its three callers `g1`, `g2`, and `g3`:

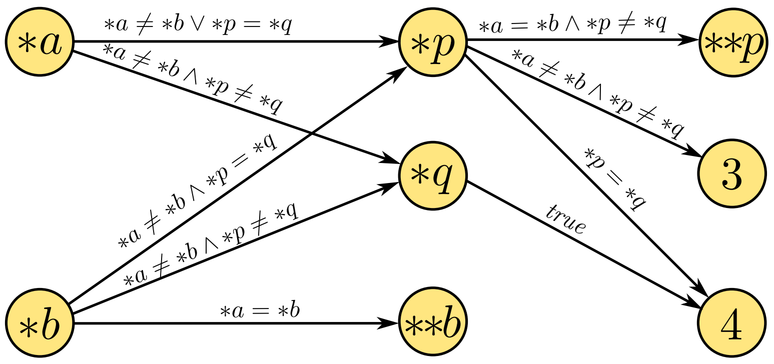

$*a$
$*p$
$**p$
$*q$
3
$*b$
$**b$
4
$*a \neq *b \vee *p = *q$
$*a \neq *b \wedge *p \neq *q$
$*a = *b \wedge *p \neq *q$
$*a \neq *b \wedge *p \neq *q$
$*p = *q$
$*a \neq *b \wedge *p = *q$
$*a \neq *b \wedge *p \neq *q$
$true$
$*a = *b$

Figure 4.1: Summary associated with function `f`

```
void f(int** a, int** b, int* p, int* q) {
  *a = p; *b = q; **a = 3; **b = 4; }
```

```
void g1() {
  int** a, int** b;
  a = new int*;
  b = new int*;
  int p = 0, q=0;
  f(a, b, &p, &q);
  assert(p == 3); }
```

```
void g2() {
  int** a, int** b;
  a = new int*;
  b = new int*;
  int p = 0;
  f(a, b, &p, &p);
  assert(p == 4); }
```

```
void g3() {
  int** a, int** b;
  a = new int*;
  b = a;
  int p = 0, q=0;
  f(a, b, &p, &q);
  assert(p == 0); }
```

Here, although the body of `f` is conditional- and loop-free, the value of `*p` after the call to `f` may be either 3, 4, or remain its initial value. In particular, in contexts where `p` and `q` are aliases (e.g., `g2`), `*p` is set to 4; in contexts where neither `a` and `b` nor `p` and `q` are aliases (e.g., `g1`), `*p` is set to 3, and in contexts where `a` and `b` are aliases but `p` and `q` are not (e.g., `g3`), the value of `*p` is unchanged after a call to `f`. Furthermore, to discharge the assertions in `g1`, `g2`, and `g3`, we need to perform strong updates to all the memory locations.

To give the reader a flavor of our technique, the function summary of `f` computed by our analysis is shown in Figure 4.1, which shows the points-to graph on exit from `f` (i.e., the heap when `f` returns). Here, points-to edges between locations are qualified by constraints, indicating the condition under which this points-to relation holds. The meaning of a constraint such as $*p = *q$ is that the location pointed to by `p` and the location pointed to by `q` are the same, i.e., `p` and `q` are aliases. Observe that Figure 4.1 encodes all possible updates to `*p` precisely: In particular, this summary indicates that `*p` has value 3 under constraint $*a \neq *b \wedge *p \neq *q$ (i.e., neither `a` and `b` nor `p` and `q` are aliases); `*p` has value 4 if `p` and `q` are aliases, and `*p` retains its initial value (`**p`) otherwise.

There are three main insights underlying our approach:

- First, we observe that a heap abstraction H at any call site of `f` can be over-approximated as the finite union of some structurally distinct *skeletal points-to graphs* $\hat{H}_1, \ldots \hat{H}_m$ where each abstract location points-to *at most one* location. This observation yields a naive, but sound, way of performing summary-based analysis where the heap state after a call to function `f` is conditioned upon the skeletal graph at the call site.

- Second, we symbolically encode all possible skeletal heaps on entry to `f` in a *single* symbolic heap where points-to edges are qualified by constraints. This insight allows us to obtain a single, *polymorphic* heap summary valid at any call site.

- Third, we observe that using summaries to apply strong updates at call sites requires a negation operation on constraints. Since these constraints may be approximations, simultaneous reasoning about may and must information on points-to relations is necessary for applying strong updates when safe. To solve this difficulty, we use *bracketing constraints* [32].

The first insight, developed in Section 4.1, forms the basic framework for reasoning about the correctness and precision of our approach. The second and third insights, exploited in Section 4.3, yield a symbolic and efficient encoding of the basic approach. To summarize, this chapter makes the following contributions:

- We develop a theory of abstract heap decompositions that elucidates the basic principle underlying modular heap analyses. This theory shows that a summary-based analysis must lose extra precision over a non-summary based analysis in some circumstances and also sheds light on the correctness of earlier work on modular alias analyses, such as [58, 79, 24].

- We present a full algorithm for performing modular heap analysis in a symbolic and efficient way. While our algorithm builds on the work of [24] in predicating summaries on aliasing patterns, our approach is much more precise and is capable of performing strong updates to heap locations at call sites.
- We demonstrate experimentally that our approach is both scalable and precise for verifying properties about real C and C++ applications up to 100,000 lines of code.

4.1 Foundations of Modular Heap Analysis

As mentioned earlier, our goal is to analyze a function f independently of its callers and generate a summary valid in any context. The main difficulty for such an analysis is that f's *heap fragment* (the portion of the program's heap reachable through f's arguments and global variables on entry to f) is unknown and may be arbitrarily complex, but a modular analysis must model this unknown heap fragment in a conservative way.

Our technique models f's heap fragment using abstractions $H_1, \ldots, H_k$ such that (i) in each H_i, every location points to *exactly one location variable* representing the unknown points-to targets of that location on function entry, (ii) each H_i represents a distinct aliasing pattern that may arise in some calling context, and (iii) the heap fragment reachable in f at any call site is overapproximated by combining a subset of the heaps in $H_1, \ldots, H_k$.

As the above discussion illustrates, our approach requires representing the heap abstraction at any call site as the finite union of heap abstractions where each pointer location has exactly one target. We observe that every known modular heap analysis, including ours, has this this *one-target* property. In principle, one could allow the unknown locations in a function's initial heap fragment to point to 2, 3, or any number of other unknown heap locations, but it is unclear how to pick the number or take advantage of the potential extra precision.

In this section, we present *canonical decompositions*, through which the heap is decomposed into a set of heaps with the one-target property, and *structural decompositions*, which coalesce isomorphic canonical heaps. We then show how these decompositions can be used for summary-based heap analysis.

4.1.1 Preliminaries

We describe the basic ideas on a standard *may points-to* graph, which we usually call a *heap* for brevity. A labeled node A represents one or more concrete memory locations $\zeta(A)$.

Definition 13 (Summary Location) *An abstract location that may represent multiple concrete locations is a* summary location *(e.g., modeling elements in an array/list). An abstract location representing exactly one concrete location is a* non-summary location.

For any two distinct abstract locations A and A', we require $\zeta(A) \cap \zeta(A') = \emptyset$, and that $|\zeta(A)| = 1$ if A is a non-summary node. An edge (A, B) in the points-to graph denotes a partial function $\zeta_{(A,B)}$ from pointer locations in $\zeta(A)$ to locations in $\zeta(B)$, with the requirement that for every pointer location $l \in \zeta(A)$ there is exactly one node B such that $\zeta_{(A,B)}(l)$ is defined (i.e., each pointer location has a unique target in a concrete heap). Finally, each possible choice of ζ and compatible edge functions $\zeta_{(A,B)}$ for each edge (A, B) maps a points-to graph H to one concrete heap. We write $\gamma(H)$ for the set of all such possible concrete heaps for the points-to graph H. We also write $H_1 \sqsupseteq H_2$ if $\gamma(H_1) \supseteq \gamma(H_2)$, and $H_1 \sqcup H_2$ for the heap that is the union of all nodes and edges in H_1 and H_2. We define a semantic judgment $H \models S : H'$ as:

$$H \models S : H' \Leftrightarrow \forall h \in \gamma(H).\ \exists h' \in \gamma(H').\ \mathit{eval}(h, S) = h'$$

where $\mathit{eval}(h, S)$ is the result of executing code fragment S starting with concrete heap h. Now, we write $H \vdash_a S : H'$ to indicate that, given a points-to graph H and a program fragment S, H' is the new heap obtained after analyzing S using pointer

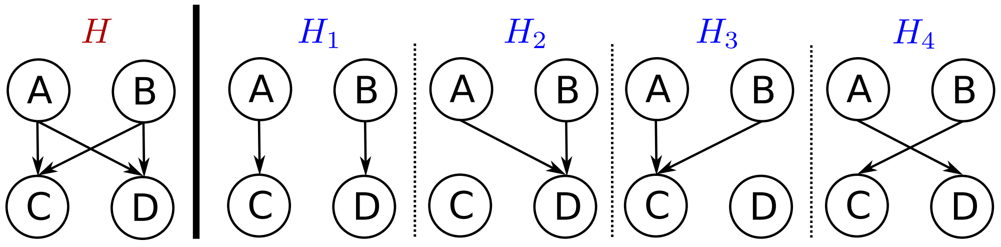


Figure 4.2: A heap H and its canonical decomposition $H_1, \ldots, H_4$

analysis a. The pointer analysis a is sound if for all program fragments S:

$$H \vdash_a S : H' \;\Rightarrow\; H \models S : H'$$

4.1.2 Canonical Decomposition

In this section, we describe how to express a points-to graph H as the union of a set of points-to graphs $H_1, \ldots, H_k$ where in each H_i, every abstract location points to *at most one* location.

Definition 14 (Canonical points-to graph) *We say a points-to graph is* canonical *if every abstract memory location has an edge to at most one abstract memory location.*

Definition 15 (Canonical decomposition) *The canonical decomposition of heap H is obtained by applying these steps in order:*

1. *If a summary node A points to multiple locations $T_1, \ldots, T_k$, replace $T_1, \ldots, T_k$ with a single summary node T such that any edge to/from any T_i is replaced with an edge to/from T.*

2. *Let B be a location with multiple edges to $T_1, \ldots, T_k$. Split the heap into $H_1, \ldots, H_k$ where in each H_i, B has exactly one edge to T_i, and recursively apply this rule to each H_i.*

Lemma 4 *Let $H_1, \ldots, H_k$ be the canonical decomposition of H. Then $(H_1 \sqcup \ldots \sqcup H_k) \sqsupseteq H$.*

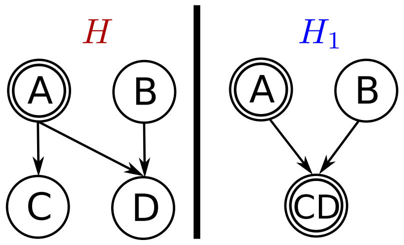


Figure 4.3: A heap H and its canonical decomposition H_1

Proof 3 *Let H' be the heap obtained from step 1 of Definition 15. To show $H' \sqsupseteq H$ we must show $\gamma(H') \supseteq \gamma(H)$. Let $h \in \gamma(H)$ and let ζ_H by the corresponding mapping. We choose $\zeta^{H'}(T) = \zeta^H(T_1) \cup ... \cup \zeta^H(T_k)$ and $\zeta^{H'}(X) = \zeta^H(X)$ otherwise, and construct the edge mappings $\zeta^H_{(A,B)}$ from $\zeta^{H'}_{(A,B)}$ analogously. Thus, $h \in \gamma(H')$ and we have $\gamma(H') \supseteq \gamma(H)$. In step 2, observe that any location B with multiple edges to $T_1, \ldots, T_k$ must be a non-summary location. Hence, the only concrete location represented by B must point to exactly one T_i in any execution. Thus, in this step, $(H_1 \sqcup \ldots \sqcup H_k) = H' \sqsupseteq H$.* $\Box$

Example 17 *Figure 4.2 shows a heap H with only non-summary locations. The canonical decomposition of H is H_1, H_2, H_3, H_4, representing four different concrete heaps encoded by H.*

Example 18 *Figure 4.3 shows another heap H with summary node A (indicated by double circles) and its canonical decomposition H_1. Heap H_1 is obtained from H by collapsing locations C and D into a summary location CD. Observe that we cannot split H into two heaps H_1 and H_2 where A points to C in H_1 and to D in H_2: Such a decomposition would incorrectly state that all elements in A must point to the same location, whereas H allows distinct concrete elements in A to point to distinct locations.*

Corollary 1 *If H has no summary nodes with multiple edges, then its canonical decomposition is exact, i.e., $\bigsqcup_{1 \leq i \leq k} H_i = H$.*

Proof 4 *This follows immediately from the proof of Lemma 4.* □

Lemma 5 *Consider a sound pointer analysis "a" and a heap H with canonical decomposition $H_1, \ldots, H_k$ such that:*

$$H_1 \vdash_a S : H'_1 \quad \ldots \quad H_k \vdash_a S : H'_k$$

Then, $H \models S : H'_1 \sqcup \ldots \sqcup H'_k$.

Proof 5 *This follows directly from Lemma 4.* □

According to this lemma, we can conservatively analyze a program fragment S by first decomposing a heap H into canonical heaps $H_1, \ldots, H_k$, then analyzing S using each initial heap H_i, and finally combining the resulting heaps $H'_1, \ldots, H'_k$.

Recall that in a modular heap analysis, we require each node in a function f's initial heap abstraction to have the single-target property. Corollary 1 implies that if a call site of f has no summary nodes with multiple targets, then this assumption results in no loss of information, because we can use multiple distinct heaps for f that, in combination, are an exact representation of the call site's heap. However, if a summary location has multiple targets and there is aliasing involving that summary node, as illustrated in Figure 4.3, the modular analysis may strictly overapproximate the heap after a call to f. In this case, the requirement that f's initial heap have the single-target property means that f can only represent the call-site's heap (shown on the left of Figure 4.3) by an overapproximating heap that merges the target nodes (shown on the right of Figure 4.3).

4.1.3 Structural Decomposition

Consider the result of analyzing a program fragment S starting with initial canonical heaps H_1 and H_2 shown in Figure 4.4. Here, nodes labeled X and Y represent memory locations of `x` and `y`, which are the only variables in scope in S. Since the only difference between H_1 and H_2 is the label of the node pointed to by `x` and `y`, the heaps H'_1 and H'_2 obtained after analyzing S will be identical except for the label of a

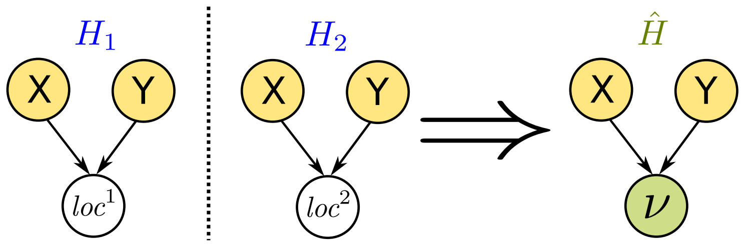


Figure 4.4: Two isomorphic canonical heaps and their skeleton

single node. Thus, S can be analyzed only once starting with heap $\hat{H}$ in Figure 4.4, and H_1' and H_2' can be obtained from the resulting heap by renaming ν to loc^1 and loc^2 respectively. The rest of this section makes this discussion precise.

Definition 16 (Skeleton) *Given a set of nodes N, let $\xi_N(H)$ be the heap obtained by erasing the labels of all nodes in H except for those in N. Now ξ_N defines an equivalence relation $H \equiv_N H'$ if $\xi_N(H) = \xi_N(H')$. We select one heap in each equivalence class of $\equiv_N$ as the class' unique* skeleton*.*

Note that nodes of skeletons are labeled—label erasure is only used to determine equivalence class membership.

Example 19 *In Figure 4.4, H_1 and H_2 have the same skeleton $\hat{H}$.*

In other words, if heaps $H_1, \ldots, H_k$ have the same aliasing patterns with respect to a set of root locations N, then $\hat{H}$ is a unique points-to graph which represents their common aliasing structure. Skeletons are useful because, if N represents formals and globals in a function f, all possible aliasing patterns at call sites of f can be expressed using a finite number of skeletons.

Definition 17 (Π) *Let H be a heap and let $\hat{H}$ be its skeleton w.r.t. nodes N. The mapping $\Pi_{H,\hat{H}}$ maps every node label in $\hat{H}$ to the label of the corresponding node in H and any other node to itself.*

Definition 18 (Structural Decomposition) *Given heap H and nodes N, the* structural decomposition *of H w.r.t. N is a set of heaps D such that for every H_i in the canonical decomposition of H, the sketelon $\hat{H}$ of H_i w.r.t. N is in D.*

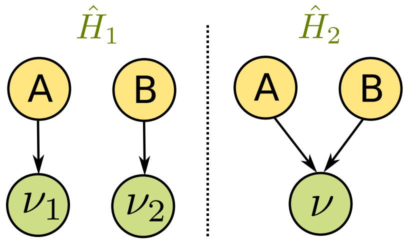


Figure 4.5: Structural decomposition of heap H from Figure 4.2

Observe that the cardinality of the structural decomposition of H is never larger than the cardinality of H's canonical decomposition.

Definition 19 (Instances of skeleton) *Let $\hat{H}$ be a skeleton in the structural decomposition of H. The* instances *of $\hat{H}$, written $\mathcal{I}_H(\hat{H})$, are the canonical heaps of H with skeleton $\hat{H}$.*

Example 20 *Consider heap H from Figure 4.2 and the root set $\{A, B\}$. The structural decomposition $\hat{H}_1, \hat{H}_2$ of H is shown in Figure 4.5. Observe that canonical heaps H_1 and H_4 from Figure 4.2 have the same skeleton $\hat{H}_1$, and H_2 and H_3 have skeleton $\hat{H}_2$. Thus, $\mathcal{I}_H(\hat{H}_1) = \{H_1, H_4\}$ and $\mathcal{I}_H(\hat{H}_2) = \{H_2, H_3\}$. Also:*

$$\begin{array}{ll} \Pi_{H_1,\hat{H}_1} = [\nu_1 \mapsto C, \nu_2 \mapsto D] & \Pi_{H_4,\hat{H}_1} = [\nu_1 \mapsto D, \nu_2, \mapsto C] \\ \Pi_{H_2,\hat{H}_2} = [\nu \mapsto D] & \Pi_{H_3,\hat{H}_2} = [\nu \mapsto C] \end{array}$$

Lemma 6 *Consider program fragment S and nodes N representing variables in scope at S. Let H_N be the heap fragment reachable through N before analyzing S and let $\hat{H}_1, \ldots, \hat{H}_m$ be the structural decomposition of H_N w.r.t. N. If*

$$\hat{H}_1 \vdash_a S : \hat{H}'_1 \quad \ldots \quad \hat{H}_m \vdash_a S : \hat{H}'_m$$

and if $\hat{H}'_N$ is the heap defined as:

$$\hat{H}'_N = \bigsqcup_{1 \leq i \leq m} \left(\bigsqcup_{H_{ij} \in \mathcal{I}_{H_N}(\hat{H}_i)} \Pi_{H_{ij},\hat{H}_i}(\hat{H}'_i) \right)$$

then $H_N \models S : \hat{H}'_N$.

Proof 6 *First, by Definitions 16 and 14, we have:*

$$\bigsqcup_{1\le i\le m}\left(\bigsqcup_{H_{ij}\in\mathcal{I}_{H_N}(\hat{H}_i)} H_{ij}\right) \sqsupseteq H_N$$

Second, using Lemma 5, this implies:

$$H_N \models S : \bigsqcup_{1\le i\le m}\left(\bigsqcup_{H_{ij}\in\mathcal{I}_{H_N}(\hat{H}_i)} H'_{ij}\right) \quad (*)$$

where $H_{ij} \vdash_a S : H'_{ij}$. *From Definition 19, since* H_{ij} *and* $\hat{H}_i$ *are equivalent up to renaming, then* H'_{ij} *and* $\hat{H}'_i$ *are also equivalent up to this renaming, given by* $\Pi_{H_{ij},\hat{H}_i}$. *Together with (*), this implies* $H_N \models S : \bigsqcup_{1\le i\le m}\left(\bigsqcup_{H_{ij}\in\mathcal{I}_{H_N}(\hat{H}_i)} \Pi_{H_{ij},\hat{H}_i}(\hat{H}'_i)\right)$. $\square$

In other words, the heap defined as $\hat{H}'_N$ in Lemma 6 gives us a sound abstraction of the heap after analyzing program fragment S. Furthermore, $\hat{H}'_N$ is precise in the sense defined below:

Lemma 7 *Let* $\hat{H}'_N$ *be the heap defined in Lemma 6, and let* $H_1, \ldots, H_k$ *be the canonical decomposition of the heap fragment reachable from* N *before analyzing* S. *If* $H_j \vdash_a H'_j$, *then:*

$$\hat{H}'_N = \bigsqcup_{1\le j\le k} H'_j$$

Proof 7 *This follows from Corollary 1 and Definition 18.* $\square$

The following corollary states a stronger precision result:

Corollary 2 *Let* H_N *and* $\hat{H}'_N$ *be the heap abstractions from Lemma 6, and let* $H_N \vdash_a S : H'_N$. *If* H_N *does not contain summary locations with multiple points-to targets, then*

$$\hat{H}'_N \sqsubseteq H'_N$$

Proof 8 *This follows from Lemma 7 and Corollary 1.*

4.1.4 From Decompositions to Modular Heap Analysis

We now show how the ideas described so far yield a basic modular heap analysis. In the remainder of this section, we assume there is a fixed bound on the total number of memory locations used by a program analysis. (In practice, this is achieved by, e.g., collapsing recursive fields of data structures to a single summary location.)

Lemma 8 *Consider a function f, and let N denote the abstract memory locations associated with the formals and globals of f. Then, there is a finite set Q of skeletons such that the structural decomposition D w.r.t. N of the heap fragment reachable from N in any of f's call sites satisfies $D \subseteq Q$.*

Proof 9 *Recall that in any canonical heap, every location has exactly one target. Second, observe that when there is bound b on the total number of locations in any heap, any canonical heap must have at most b locations. Thus, using a fixed set of nodes, we can only construct a finite set Q of structurally distinct graphs.* $\square$

Since there are a bounded number of skeletons that can arise in any context, this suggests the following strategy for computing a complete summary of function f: Let N be the set of root locations (i.e., formals and globals) on entry to f, and let $\hat{H}_1, \ldots, \hat{H}_k$ be the set of all skeletons that can be constructed from root set N. We analyze f's body for each initial skeleton $\hat{H}_i$, obtaining a new heap $\hat{H}_i'$. Now, let C be a call site of f and let R be the subset of the skeletons $\hat{H}_1, \ldots, \hat{H}_k$ that occur in the structural decomposition of heap H in context C. Then, following Lemma 6, the heap fragment after the call to f can be obtained as:

$$\bigsqcup_{\hat{H}_i \in R} \left(\bigsqcup_{H_{ij} \in \mathcal{I}_H(\hat{H}_i))} \Pi_{H_{ij}, \hat{H}_i}(\hat{H}_i') \right)$$

This strategy yields a fully context-sensitive analysis because f's body is analyzed for any possible entry aliasing pattern $\hat{H}_i$, and at a given call site C, we only use the resulting heap $\hat{H}_i$ if $\hat{H}_i$ is part of the structural decomposition of the heap at C.

Furthermore, as indicated by Corollary 2, this strategy is as precise as analyzing the inlined body of the function if there are no summary locations with multiple points-to targets at this call site; otherwise, the precision guarantee is stated in Lemma 7.

4.1.5 Discussion

While the decompositions described here are useful for understanding the principle underlying modular heap analyses, the naive algorithm sketched in Section 4.1.4 is completely impractical for two reasons: First, since the number of skeletons may be exponential in the number of abstract locations reachable through arguments, such an algorithm requires analyzing a function body exponentially many times. Second, although two initial skeletons may be different, the resulting heaps after analyzing the function body may still be identical. In the rest of this chapter, we describe a symbolic encoding of the basic algorithm that does not analyze a function more than once unless cycles are present in the callgraph (see Section 4.3). Then, in Section 4.4, we show how to identify only those initial skeletons that may affect the heap abstraction after the function call.

4.2 Language

To formalize our symbolic algorithm for modular heap analysis, we use the following typed, call-by-value imperative language:

$$\begin{array}{ll}
\textit{Program } P & := F^{+} \\
\textit{Function } F & := \textit{define } f(a_1 : \tau_1, \ldots, a_n : \tau_n) = S; \\
\textit{Statement } S & := v_1 \leftarrow *v_2 \mid\ *v_1 \leftarrow v_2 \mid v \leftarrow \textit{alloc}^{\rho}(\tau) \\
 & \mid f^{\rho}(v_1, \ldots, v_k) \mid \textit{assert}(v_1 = v_2) \\
 & \mid \textit{let}^{\rho}\ v : \tau\ \textit{in}\ S\ \textit{end} \mid S_1; S_2 \mid \textit{choose}(S_1, S_2) \\
\textit{Type } \tau & := \ \textit{int} \mid \textit{ptr}(\tau)
\end{array}$$

A program P consists of one or more (possibly recursive) functions F. Statements in this language are loads, stores, memory allocations, function calls, assertions, let bindings, sequencing, and the *choose* statement, which non-deterministically executes either S_1 or S_2 (i.e., a simplified conditional). Allocations, function calls, and let bindings are labeled with globally unique program points ρ.

Since this language is standard, we omit its operational semantics and highlight a few important assumptions: Execution starts at the first function defined, and an assertion failure aborts execution. Also, all bindings in the concrete store have initial value *nil*.

4.3 Modular & Symbolic Heap Analysis

In this section, we formally describe our symbolic algorithm for modular heap analysis. In Section 4.3.1, we first decribe the abstract domain used in our analysis. Section 4.3.2 formally defines *function summaries*, and Section 4.3.3 presents a full algorithm for summary-based heap analysis for the language defined in Section 4.2.

4.3.1 Abstract Domain

Abstract locations π represent a set of concrete locations:

$$\begin{array}{rcl} \textit{Abstract Locations } \pi & := & \alpha \mid l \\ \textit{Location Variables } \alpha & := & \nu_i \mid *\alpha \\ \textit{Location Constants } l & := & loc^{\vec{\rho}} \mid nil \end{array}$$

An abstract location π in function f is either a *location variable* α or a *location constant* l. Location constants represent stack or heap allocations in f and its transitive callees as well as *nil*. In contrast, location variables represent the unknown memory locations reachable from f's arguments at call sites, similar to access paths in [58]. Informally, location variables correspond to the node labels of a skeleton from Section 4.1.3. Recall from Section 4.1 that, in any canonical points-to graph, every abstract memory location points to at most one other abstract memory location; hence, location variable $*\nu_i$ describes the unknown, but unique, points-to target of

f's i'th argument in some canonical heap at a call site of f.

Abstract environment $\mathbb{E}$ maps program variables v to abstract locations π, and abstract store $\mathbb{S}$ maps each abstract location π to an *abstract value set* θ of (abstract location, constraint) pairs:

$$\textit{Abstract value set } \theta := 2^{(\pi,\phi)}$$

The abstract store defines the edges of the points-to graph from Section 4.1. A mapping from abstract location π to abstract value set $\{(\pi_1, \phi_1), \ldots, (\pi_k, \phi_k)\}$ in $\mathbb{S}$ indicates that the heap abstraction contains a points-to edge from node labeled π to nodes labeled $\pi_1, \ldots, \pi_k$. Observe that, unlike the simple may points-to graph we considered in Section 4.1, points-to edges in the abstract store are qualified by constraints, which we utilize to symbolically encode all possible skeletons in one symbolic heap (see Section 4.3.3).

Constraints in our abstract domain are defined as follows:

$$\begin{array}{rcl} \phi & := & \langle \varphi_{may}, \varphi_{must} \rangle \\ \varphi & := & T \mid F \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \neg\varphi \mid t_1 = t_2 \end{array}$$

Here, ϕ is a *bracketing constraint* $\langle \varphi_{may}, \varphi_{must} \rangle$ as in [32], representing the condition under which a property may and must hold. Recall that the simultaneous use of may and must information is necessary for applying strong updates whenever safe. In particular, updates to heap locations require negation (see Section 4.3.3). Since the negation of an overapproximation is an underapproximation, the use of bracketing constraints allows a sound negation operation, defined as $\neg\langle \varphi_{may}, \varphi_{must} \rangle = \langle \neg\varphi_{must}, \neg\varphi_{may} \rangle$. Conjunction and disjunction are defined on these constraints as expected:

$$\langle \varphi_{may}, \varphi_{must} \rangle \star \langle \varphi'_{may}, \varphi'_{must} \rangle = \langle \varphi_{may} \star \varphi'_{may}, \varphi_{must} \star \varphi'_{must} \rangle$$

where $\star \in \{\wedge, \vee\}$. In this chapter, any constraint ϕ is a bracketing constraint unless stated otherwise. To make this clear, any time we do not use a bracketing constraint,

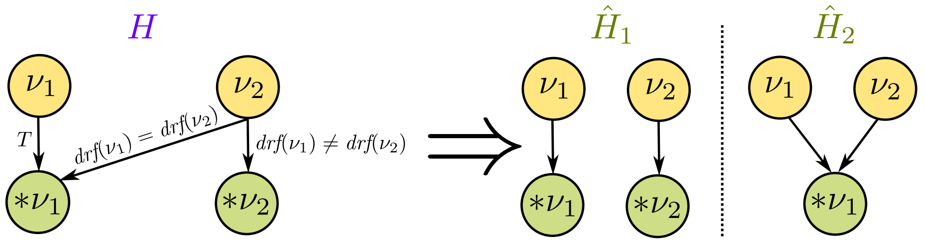


Figure 4.6: A symbolic heap representing two skeletons

we use the letter φ instead of ϕ. Furthermore, if the may and must conditions of a bracketing constraint are the same, we write a single constraint instead of a pair. Finally, for a bracketing constraint $\phi = \langle \varphi_{may}, \varphi_{must} \rangle$, we define $\lceil \phi \rceil = \varphi_{may}$ and $\lfloor \phi \rfloor = \varphi_{must}$.

In the definition of constraint φ, T and F represent the boolean constants true and false, and a term t is defined as:

$$Term\ t := v \mid drf(t) \mid alloc(\vec{\rho}) \mid nil$$

Here, v represents a variable, *drf* is an uninterpreted function, and *alloc* is an *invertible* uninterpreted function applied to a vector of constants $\vec{\rho}$. Thus, constraints φ belong to the theory of equality with uninterpreted functions. Our analysis requires converting between abstract locations and terms in the constraint language; we therefore define a *lift* operation, written $\overline{\pi}$, for this purpose:

$$\overline{\nu_i} = \nu_i \quad \overline{*\alpha} = drf(\overline{\alpha}) \quad \overline{nil} = nil \quad \overline{loc^{\vec{\rho}}} = alloc(\vec{\rho})$$

Observe that a location constant loc^{ρ} is converted to a term $alloc(\vec{\rho})$, which effectively behaves as a constant in the constraint language: Since *alloc* is an invertible function, $alloc(\vec{\rho}) = alloc(\vec{\rho'})$ exactly when $\rho = \rho'$. A location variable ν is converted to a constraint variable of the same name, and the location variable $*\nu$ is represented by the term $drf(\nu)$ which represents the unknown points-to target of ν on function entry. We write $lift^{-1}(t)$ to denote the conversion of a term to an abstract location.

Example 21 *In Figure 4.6, a symbolic heap H represents two skeletons $\hat{H}_1$ and $\hat{H}_2$.*

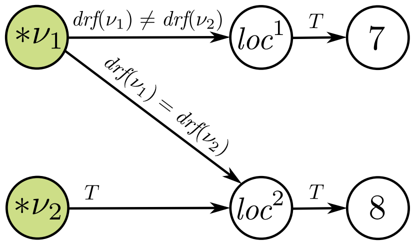


Figure 4.7: The abstract store in f's summary

In H, the constraint $\text{drf}(\nu_1) = \text{drf}(\nu_2)$ *describes contexts where the first and second arguments are aliases. Observe that, at call sites where the first and second arguments alias,* $\text{drf}(\nu_1) = \text{drf}(\nu_2)$ *instantiates to* true *and* $\text{drf}(\nu_1) \neq \text{drf}(\nu_2)$ *is* false*; thus at this call site, H instantiates exactly to $\hat{H}_2$. Similarly, if the first and second arguments do not alias, H instantiates to $\hat{H}_1$.*

4.3.2 Function Summaries

A *summary* Δ for a function f is a pair $\Delta = \langle \phi, \mathbb{S} \rangle$ where ϕ is a constraint describing the precondition for f to succeed (i.e., not abort), and $\mathbb{S}$ is a symbolic heap representing the heap abstraction after f returns. More specifically, let $\hat{H}_1, \ldots, \hat{H}_k$ be the set of all skeletons for any possible call site of f, and let $\hat{H}_i \vdash S : \hat{H}'_i$ where S is the body of f. Then, the abstract store $\mathbb{S}$ symbolically encodes that in contexts where the constraints in $\mathbb{S}$ are satisfied by initial heap $\hat{H}_i$, the resulting heap is $\hat{H}'_i$.

Observe that a summary can also be viewed as the Hoare triple $\{\phi\}\ f\ \{\mathbb{S}\}$. Thus, the computation of a summary for f is equivalent to the inference of sound pre- and post-conditions for f.

Example 22 *Consider the function:*

```
define $\texttt{f}(\texttt{a}_1 : \texttt{ptr(ptr(int))},\ \texttt{a}_2 : \texttt{ptr(ptr(int))}) =$
1 :   $*\texttt{a}_1 \leftarrow \texttt{alloc}^1\texttt{(int)};$
2 :   $*\texttt{a}_2 \leftarrow \texttt{alloc}^2\texttt{(int)};$
3 :   let $\texttt{t}_1 : \texttt{ptr(int)}$ in $\texttt{t}_1 \leftarrow *\texttt{a}_1;\ *\texttt{t}_1 \leftarrow 7$ end;
4 :   let $\texttt{t}_2 : \texttt{ptr(int)}$ in $\texttt{t}_2 \leftarrow *\texttt{a}_2;\ *\texttt{t}_2 \leftarrow 8$ end;
5 :   let $\texttt{t}_3 : \texttt{ptr(int)}$ in $\texttt{t}_3 \leftarrow *\texttt{a}_1;$
6 :        let $\texttt{t}_4 : \texttt{int}$ in $\texttt{t}_4 \leftarrow *\texttt{t}_3;\ \texttt{assert}(\texttt{t}_4 == 7)$ end;
7 :   end
```

The summary for f *is* $\langle\ \mathrm{drf}(\nu_1) \neq \mathrm{drf}(\nu_2),\ \mathbb{S}\ \rangle$ *where* $\mathbb{S}$ *is shown in Figure 4.7. The pre-condition* $\mathrm{drf}(\nu_1) \neq \mathrm{drf}(\nu_2)$ *indicates that the assertion fails in those contexts where arguments of* f *are aliases. Also, in symbolic heap* $\mathbb{S}$*, the abstract location reached by dereferencing* a_1 *(whose location is* ν_1*) is either* loc_1*, corresponding to the allocation at line 1, or* loc_2*, associated with the allocation at line 2, depending on whether* a_1 *and* a_2 *are aliases.*

A *global summary environment* $\mathbb{G}$ is a mapping from each function f in the program to a summary Δ_f.

4.3.3 The Analysis

We now present the full algorithm for the language of Section 4.2. Section 4.3.3 describes the symbolic initialization of the local heap to account for all possible aliasing relations on function entry. Section 4.3.3 gives abstract transformers for all statements except function calls, which is described in Section 4.3.3. Finally, Section 4.3.3 describes the generation of function summaries.

Local Heap Initialization

To analyze a function f independent of its callers, we initialize f's abstract store to account for all possible relevant aliasing relationships at function entry. To perform this local heap initialization, we utilize an *alias partition environment* $\mathbb{A}$ with the signature $\alpha \to 2^{\alpha}$. This environment maps each location variable α to an *ordered* set

$$\frac{\begin{array}{l}\mathbb{E} = [a_1 \leftarrow \nu_1, \ldots, a_k \leftarrow \nu_k] \\ \forall \;\; \alpha_i \in dom(\mathbb{A}). \\ \quad \mathbb{S}(\alpha_i) \leftarrow \cup_{k \le i}(*\alpha_k, (\bigwedge_{j<k} \overline{*\alpha_i} \neq \overline{*\alpha_j}) \wedge \overline{*\alpha_i} = \overline{*\alpha_k})\end{array}}{\mathbb{A} \vdash \textit{init\_heap}(a_1, \ldots, a_k) : \mathbb{E}, \mathbb{S}}$$

Figure 4.8: Local Heap Initialization

of location variables, called α's *alias partition set.* If $\alpha' \in \mathbb{A}(\alpha)$, then f's summary may differ in contexts where α and α' are aliases. Since aliasing is a symmetric property, any alias partition environment $\mathbb{A}$ has the property $\alpha' \in \mathbb{A}(\alpha) \Leftrightarrow \alpha \in \mathbb{A}(\alpha')$. Any location aliases itself, and so $\mathbb{A}$ is also reflexive: $\alpha \in \mathbb{A}(\alpha)$. A correct alias partition environment $\mathbb{A}$ can be trivially computed by stipulating that $\alpha' \in \mathbb{A}(\alpha)$ if α and α' have the same type. We discuss how to compute a more precise alias partition environment $\mathbb{A}$ in Section 4.4.

A key component of the modular analysis is the *init_heap* rule in Figure 4.8. Given formal parameters $a_1, \ldots, a_k$ to function f, this rule initializes the abstract environment and store on entry to f. The environment $\mathbb{E}$ is initialized by binding a location variable ν_i to each argument a_i. The initialization of the abstract store $\mathbb{S}$, however, is more involved because we need to account for all possible entry aliasing relationships permitted by $\mathbb{A}$.

Intuitively, if $\mathbb{A}$ indicates that α_1 and α_2 may alias on function entry, we need to analyze f's body with two skeletal heaps, one where α_1 and α_2 point to the same location, and one where α_1 and α_2 point to different locations. To encode this symbolically, one obvious solution is to introduce three location variables, $*\alpha_1$, $*\alpha_2$, and $*\alpha_{12}$ such that α_1 (resp. α_2) points to $*\alpha_1$ (resp. $*\alpha_2$) if they do not alias (i.e., under constraint $\textit{drf}(\alpha_1) \neq \textit{drf}(\alpha_2)$) and point to a common location named $*\alpha_{12}$ if they alias (i.e., under $\textit{drf}(\alpha_1) = \textit{drf}(\alpha_2)$). While this encoding correctly describes both skeletal heaps, it unfortunately introduces an exponential number of locations, one for each subset of entry alias relations in $\mathbb{A}$.

To avoid this exponential blow-up, we impose a total order on abstract locations such that if α_i and α_j are aliases, they both point to a common location $*\alpha_k$ such that α_k is the least element in the alias partition class of α_i and α_j. Thus, in the

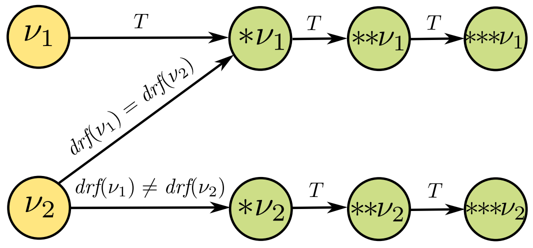


Figure 4.9: The initial heap abstraction for function from Example 22

init_heap rule of Figure 4.8, α_i points to $*\alpha_k$ where $k \leq i$ under constraint:

$$(\bigwedge_{j<k} \overline{*\alpha_i} \neq \overline{*\alpha_j}) \quad \wedge \quad \overline{*\alpha_i} = \overline{*\alpha_k}$$

This condition ensures that α_i points to a location named $*\alpha_k$ only if it does not alias any other location $\alpha_j \in \mathbb{A}(\alpha_i)$ with $j < k$.

Example 23 *Consider the function defined in Example 22. Suppose the alias partition environment $\mathbb{A}$ contains the following mappings:*

$$\nu_1 \mapsto \{\nu_1, \nu_2\}, \nu_2 \mapsto \{\nu_1, \nu_2\}, *\nu_1 \mapsto \{*\nu_1\}, *\nu_2 \mapsto \{*\nu_2\},$$
$$** \nu_1 \mapsto \{** \nu_1\}, ** \nu_2 \mapsto \{** \nu_2\}$$

Figure 4.9 shows the initial heap abstraction using $\mathbb{A}$ and the ordering $\nu_1 < \nu_2$. Since $\mathbb{A}$ includes ν_1 and ν_2 in the same alias partition set, ν_2 points to $*\nu_1$ under $\text{drf}(\nu_1) = \text{drf}(\nu_2)$ *and to $*\nu_2$ under its negation. But ν_1 only points to $*\nu_1$ since $\nu_2 \not< \nu_1$.*

The following lemma states that the initial heap abstraction correctly accounts for all entry aliasing relations permitted by $\mathbb{A}$:

Lemma 9 *Let α_i and α_j be two abstract locations such that $\alpha_j \in \mathbb{A}(\alpha_i)$. The initial local heap abstraction $\mathbb{S}$ constructed in Figure 4.8 encodes that α_i and α_j point to distinct locations* exactly *in those contexts where they do not alias.*

Proof 10 *Without loss of generality, assume $i < j$.*
$\Rightarrow$ *Suppose α_i and α_j are not aliases in a context C, but $\mathbb{S}$ encodes they may point to*

the same location $*\alpha_k$ *in context* C*. Let* ϕ *and* ϕ' *be the constraints under which* α_i *and* α_j *point to* α_k *respectively. By construction,* $k \leq i$*, and* ϕ *implies* $\mathrm{drf}(\alpha_i) = \mathrm{drf}(\alpha_k)$ *and* ϕ' *implies* $\mathrm{drf}(\alpha_j) = \mathrm{drf}(\alpha_k)$*. Thus, we have* $\mathrm{drf}(\alpha_i) = \mathrm{drf}(\alpha_j)$*, contradicting the fact that* α_i *and* α_j *do not alias in* C*.*
$\Leftarrow$ *Suppose* α_i *and* α_j *are aliases in context* C*, but* $\mathbb{S}$ *allows* α_i *and* α_j *to point to distinct locations* $*\alpha_k$ *and* $*\alpha_m$*. Let* ϕ *and* ϕ' *be the constraints under which* α_i *points to* $*\alpha_k$ *and* α_j *points to* $*\alpha_m$ *respectively. Case (i): Suppose* $k < m$*. Then, by construction,* ϕ *implies* $\mathrm{drf}(\alpha_i) = \mathrm{drf}(\alpha_k)$*, and* ϕ' *implies* $\mathrm{drf}(\alpha_j) \neq \mathrm{drf}(\alpha_k)$*. Hence, we have* $\mathrm{drf}(\alpha_j) \neq \mathrm{drf}(\alpha_i)$*, contradicting the assumption that* α_i *and* α_j *are aliases in* C*. Case (ii):* $k > m$*. Then,* ϕ' *implies* $\mathrm{drf}(\alpha_j) = \mathrm{drf}(\alpha_m)$*, and* ϕ *implies* $\mathrm{drf}(\alpha_i) \neq \mathrm{drf}(\alpha_m)$*, again contradicting the fact that* α_i *and* α_j *are aliases in* C*.* $\square$

Lemma 10 *For each alias partition set of size* n*, the init_heap rule adds* $n(n+1)/2$ *points-to edges.*

As Lemma 10 states, this construction introduces a quadratic number of edges in the size of each alias partition set to represent all possible skeletal heaps. Furthermore, the number of abstract locations in the initial symbolic heap is no larger than the maximum number of abstract locations in any individual skeleton.

Abstract Transformers for Basic Statements

In this section, we describe the abstract transformers for all statements except function calls, which is the topic of Section 4.3.3. Statement transformers are given as inference rules of the form

$$\mathbb{E}, \mathbb{S}, \mathbb{G}, \phi \vdash S : \mathbb{S}', \phi'$$

which states that under abstract environment $\mathbb{E}$, store $\mathbb{S}$, summary environment $\mathbb{G}$, and precondition ϕ, statement S produces a new abstract store $\mathbb{S}'$ and a new precondition ϕ' of the current function. The operation $\mathbb{S}(\theta)$ looks up the value of each π_i in θ:

$$S(\{(\pi_1, \phi_1), \ldots, (\pi_k, \phi_k)\}) = \bigcup_{1 \leq i \leq k} \mathbb{S}(\pi_i) \wedge \phi_i$$

$$(1)\quad \frac{\mathbb{E}(v_1) = \pi_1 \;\; \mathbb{E}(v_2) = \pi_2 \qquad \mathbb{S}(\pi_2) = \theta \quad \mathbb{S}' = \mathbb{S}[\pi_1 \leftarrow \mathbb{S}(\theta)]}{\mathbb{E}, \mathbb{S}, \mathbb{G}, \phi \vdash v_1 \leftarrow *v_2 : \mathbb{S}', \phi}$$

$$(2)\quad \frac{\mathbb{E}(v_1) = \pi_1 \;\; \mathbb{E}(v_2) = \pi_2 \qquad \mathbb{S}(\pi_1) = \theta_1 \;\; \mathbb{S}(\pi_2) = \theta_2 \qquad \mathbb{S}' = \mathbb{S}[\pi_i \leftarrow ((\theta_2 \wedge \phi_i) \cup (\mathbb{S}(\pi_i) \wedge \neg\phi_i)) \mid (\pi_i, \phi_i) \in \theta_1]}{\mathbb{E}, \mathbb{S}, \mathbb{G}, \phi \vdash *v_1 \leftarrow v_2 : \mathbb{S}', \phi}$$

$$(3)\quad \frac{\mathbb{E}(v) = \pi \qquad \mathbb{S}' = \mathbb{S}[\pi \leftarrow \{(loc^\rho, T)\},\; loc^\rho \leftarrow \{(nil, T)\}]}{\mathbb{E}, \mathbb{S}, \mathbb{G}, \phi \vdash v \leftarrow alloc^\rho(\tau) : \mathbb{S}', \phi}$$

$$(4)\quad \frac{\mathbb{E}(v_1) = \pi \;\; \mathbb{E}(v_2) = \pi' \qquad \mathbb{S}(\pi) = \{\ldots, (\pi_i, \phi_i), \ldots\} \qquad \mathbb{S}(\pi') = \{\ldots, (\pi'_j, \phi'_j), \ldots\} \qquad \phi' = \bigvee_{i,j}(\pi_i = \pi'_j \wedge \phi_i \wedge \phi'_j)}{\mathbb{E}, \mathbb{S}, \mathbb{G}, \phi \vdash assert(v_1 = v_2) : \mathbb{S}, \phi \wedge \phi'}$$

$$(5)\quad \frac{\mathbb{E}' = \mathbb{E}[v \leftarrow loc^\rho] \qquad \mathbb{S}' = \mathbb{S}[*loc^\rho \leftarrow \{(nil, T)\}] \qquad \mathbb{E}', \mathbb{S}', \mathbb{G}, \phi \vdash S : \mathbb{S}'', \phi'}{\mathbb{E}, \mathbb{S}, \mathbb{G}, \phi \vdash let^\rho\ v : \tau\ in\ S\ end\ : \mathbb{S}'' \backslash loc^\rho, \phi'}$$

$$(6)\quad \frac{\mathbb{E}, \mathbb{S}, \mathbb{G}, \phi \vdash S_1 : \mathbb{S}', \phi' \qquad \mathbb{E}, \mathbb{S}', \mathbb{G}, \phi' \vdash S_2 : \mathbb{S}'', \phi''}{\mathbb{E}, \mathbb{S}, \mathbb{G}, \phi \vdash S_1; S_2 : \mathbb{S}'', \phi''}$$

$$(7)\quad \frac{\mathbb{E}, \mathbb{S}, \mathbb{G}, \phi \vdash S_1 : \mathbb{S}_1, \phi_1 \qquad \mathbb{E}, \mathbb{S}, \mathbb{G}, \phi \vdash S_2 : \mathbb{S}_2, \phi_2}{\mathbb{E}, \mathbb{S}, \mathbb{G}, \phi \vdash\ choose\ (S_1, S_2) : \mathbb{S}_1 \sqcup \mathbb{S}_2, \phi_1 \wedge \phi_2}$$

Figure 4.10: Abstract Transformers for Basic Statements

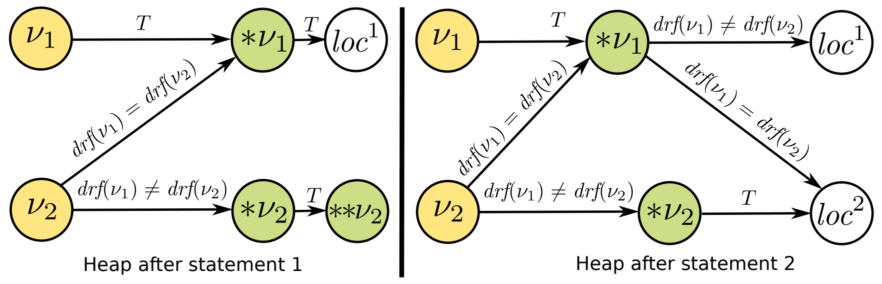

ν_1
T
$*\nu_1$
T
loc^1
$drf(\nu_1) = drf(\nu_2)$
ν_2
$drf(\nu_1) \neq drf(\nu_2)$
$*\nu_2$
T
$**\nu_2$
Heap after statement 1
ν_1
T
$*\nu_1$
$drf(\nu_1) \neq drf(\nu_2)$
loc^1
$drf(\nu_1) = drf(\nu_2)$
$drf(\nu_1) = drf(\nu_2)$
ν_2
$drf(\nu_1) \neq drf(\nu_2)$
$*\nu_2$
T
loc^2
Heap after statement 2

Figure 4.11: The symbolic heap before and after line 2 in Example 22

where $\mathbb{S}(\pi_i) \wedge \phi_i$ is a shorthand defined as follows:

$$\theta \wedge \phi = \{(\pi_j, \phi_j \wedge \phi) | (\pi_j, \phi_j) \in \theta\}$$

In Figure 4.10, rules (1) and (2) give the transformers for loads and stores respectively. The rule for loads is self-explanatory; thus, we focus on the store rule. In the third hypothesis of rule (2), each π_i represents a location that v_1 points to under constraint ϕ_i, and θ_2 is the value set for v_2. Since the write to π_i happens under constraint ϕ_i, the new value of π_i in $\mathbb{S}'$ is θ_2 under constraint ϕ_i and retains its old value set $\mathbb{S}(\pi_i)$ under $\neg\phi_i$. Observe that if ϕ_i is *true*, this rule performs a standard strong update to π_i. On the other hand, if v_1 points to π_i under some entry alias assumption, then there is a strong update to π_i exactly in those calling contexts where this alias assumption holds.

Example 24 *Figure 4.11 shows the relevant portion of the heap abstraction before and after the store at line 2 in Example 22.*

Rule (3) processes allocations by introducing a new location loc^ρ and initializing its value in the store to *nil*. Rule (4) analyzes an assertion by computing the condition ϕ' for the assertion to hold such that if ϕ' can be proven valid in a calling context, then this assertion must hold at that call site. In rule (4), ϕ' is computed as the disjunction of all pairwise equalities of the elements in the two abstract value sets associated with v_1 and v_2, i.e., a case analysis of their possible values. Rule (5) describes the abstract

$$\frac{}{\mathbb{S}, \mathbb{I} \vdash \mathit{map\_loc}(\nu : \mathit{int}) : \mathbb{I}} \qquad \frac{\mathbb{S}, \mathbb{I} \vdash \mathit{map\_loc}(*\nu : \tau) : \mathbb{I}'}{\mathbb{S}, \mathbb{I} \vdash \mathit{map\_loc}(\nu : \mathit{ptr}(\tau)) : \mathbb{I}'}$$

$$\frac{\mathbb{I}' = \mathbb{I}[*\alpha \leftarrow \mathbb{S}(\mathbb{I}(\alpha))]}{\mathbb{S}, \mathbb{I} \vdash \mathit{map\_loc}(*\alpha : \mathit{int}) : \mathbb{I}'} \quad \frac{\begin{array}{c}\mathbb{I}' = \mathbb{I}[*\alpha \leftarrow \mathbb{S}(\mathbb{I}(\alpha))] \\ \mathbb{I}' \vdash \mathit{map\_loc}(** \, \alpha : \tau) : \mathbb{I}''\end{array}}{\mathbb{S}, \mathbb{I} \vdash \mathit{map\_loc}(*\alpha : \mathit{ptr}(\tau)) : \mathbb{I}''}$$

$$\frac{\begin{array}{c}\mathbb{S}, [\nu_1 \leftarrow \{(\mathbb{E}(v_1), T)\}] \vdash \mathit{map\_loc}(\nu_1) : \mathbb{I}_1 \\ \dots \\ \mathbb{S}, \mathbb{I}_{k-1}[\nu_k \leftarrow \{(\mathbb{E}(v_k), T)\}] \vdash \mathit{map\_loc}(\nu_k) : \mathbb{I}_k\end{array}}{\mathbb{E}, \mathbb{S} \vdash \mathit{map\_args}(v_1 : \tau_1, \dots, v_k : \tau_k) : \mathbb{I}_k}$$

Figure 4.12: Rules for computing instantiation environment $\mathbb{I}$

semantics of let statements by binding variable v to a new location loc^ρ in $\mathbb{E}$. Rule (6) for sequencing is standard, and rule (7) gives the semantics of the *choose* construct, which computes the join of two abstract stores $\mathbb{S}_1$ and $\mathbb{S}_2$. To define a join operation on abstract stores, we first define *domain extension*:

Definition 20 (Domain Extension) *Let π be any binding in abstract store $\mathbb{S}'$ and let (π_i, ϕ_i) be any element of $\mathbb{S}'(\pi)$. We say an abstract store $S'' = \mathbb{S}_{\mapsto \mathbb{S}'}$ is a domain extention of $\mathbb{S}$ with respect to $\mathbb{S}'$ if the following condition holds:*

1. *If $\pi \in \mathrm{dom}(\mathbb{S}) \wedge (\pi_i, \phi_i') \in \mathbb{S}(\pi)$, then $(\pi_i, \phi_i') \in \mathbb{S}_{\mapsto \mathbb{S}'}(\pi)$.*
2. *Otherwise, $(\pi_i, \mathrm{false}) \in \mathbb{S}_{\mapsto \mathbb{S}'}(\pi)$*

Definition 21 (Join) *Let $\mathbb{S}_1' = \mathbb{S}_{1 \mapsto \mathbb{S}_2}$ and let $\mathbb{S}_2' = \mathbb{S}_{2 \mapsto \mathbb{S}_1}$. If $(\pi', \langle \varphi^1_{\mathrm{may}}, \varphi^1_{\mathrm{must}} \rangle) \in \mathbb{S}_1'(\pi)$ and $(\pi', \langle \varphi^2_{\mathrm{may}}, \varphi^2_{\mathrm{must}} \rangle) \in \mathbb{S}_2'(\pi)$, then:*

$$(\pi', \langle \varphi^1_{\mathrm{may}} \vee \varphi^2_{\mathrm{may}}, \varphi^1_{\mathrm{must}} \wedge \varphi^2_{\mathrm{must}} \rangle) \in (\mathbb{S}_1 \sqcup \mathbb{S}_2)(\pi)$$

Instantiation of Summaries

The most involved statement transformer is the one for function calls, which we describe in this subsection. Figure 4.15 gives the complete transformer for function

$$\frac{}{\mathbb{I}, \rho \vdash \mathit{inst\_loc}(\mathit{nil}) : \{(\mathit{nil}, T)\}} \quad \frac{\theta = \{(\mathit{loc}^{\rho::\vec{\rho'}}, T)\}}{\mathbb{I}, \rho \vdash \mathit{inst\_loc}(\mathit{loc}^{\vec{\rho'}}) : \theta} \ (\rho \notin \vec{\rho'})$$

$$\frac{}{\mathbb{I}, \rho \vdash \mathit{inst\_loc}(\alpha) : \mathbb{I}(\alpha)} \quad \frac{\theta = \{(\mathit{loc}^{\vec{\rho'}}, \langle T, F\rangle)\}}{\mathbb{I}, \rho \vdash \mathit{inst\_loc}(\mathit{loc}^{\vec{\rho'}}) : \theta} \ (\rho \in \vec{\rho'})$$

Figure 4.13: Rules for instantiating locations

calls, making use of the helper rules defined in Figures 4.12- 4.14, which we discuss in order.

Given the actuals $v_1, \ldots, v_k$ for a call to function f, Figure 4.12 computes the *instantiation environment* $\mathbb{I}$ with signature $\alpha \to \theta$ for this call site. This environment $\mathbb{I}$, which serves as the symbolic equivalent of the mapping Π from Section 4.1, maps location variables used in f to their corresponding locations in the current (calling) function. However, since $\mathbb{I}$ is symbolic, it produces an abstract value set $\{(\pi_1, \phi_1), \ldots, (\pi_k, \phi_k)\}$ for each α such that α instantiates to π_i in some canonical heap under constraint ϕ_i.

Figure 4.13 describes the rules for instantiating any location π used in the summary. If π is a location variable, we use environment $\mathbb{I}$ to look up its instantiation. On the other hand, if π is a location constant allocated in callee f, we need to rename this constant to distinguish allocations made at different call sites for full context-sensitivity. In general, we rename the location constant $\mathit{loc}^{\vec{\rho'}}$ by prepending to $\vec{\rho'}$ the program point ρ associated with the call site. However, in the presence of recursion, we need to avoid creating an unbounded number of location constants; thus, in Figure 4.13, we check if this allocation is created on a cyclic path in the callgraph by testing whether the current program point ρ is already in $\vec{\rho'}$. In the latter case, we do not create a new location constant but weaken the bracketing constraint associated with $\mathit{loc}^{\vec{\rho'}}$ to $\langle T, F\rangle$, which has the effect of ensuring that stores into this location only apply weak updates [32], meaning that $\mathit{loc}^{\vec{\rho'}}$ behaves as a summary location.

In addition to instantiating locations, we must also instantiate the associated constraints, which is described in Figure 4.14. In the last rule of this figure, inst_ϕ instantiates a bracketing constraint, making use of inst_φ to map the constituent may and must conditions. The $inst_\varphi$ rule derives judgments of the form $\mathbb{I}, \rho \vdash inst_\varphi(\varphi) : \varphi', \phi$, where φ' preserves the structure of φ by substituting each term t in φ with a temporary variable k and ϕ constrains the values of k.

The first rule in Figure 4.14 for instantiating a leaf $t_1 = t_2$ is the most interesting one: Here, we convert t_1 and t_2 to their corresponding memory locations using the $lift^{-1}$ operation from Section 4.3.1 and instantiate the corresponding locations using *inst_loc* to obtain abstract value sets θ_1 and θ_2. We then introduce two temporary variables k and k' representing θ_1 and θ_2 respectively, and introduce constraints ϕ and ϕ', stipulating the equality between k and θ_1 and between k' and θ_2. Observe that in the last rule of Figure 4.14, these temporary variables k and k' are removed using a *QE* procedure to eliminate existentially quantified variables.

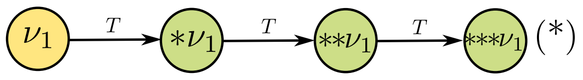
Figure 4.15 makes use of all the afore-mentioned rules to instantiate the summary of function f at a given call site ρ. In the last rule of this figure, we first look up f's summary $\langle \phi_f, \mathbb{S}_f \rangle$ in the global summary environment $\mathbb{G}$. The precondition ϕ_f is instantiated to ϕ'_f using $inst_\phi$. Observe that if ϕ'_f is valid, then the potential assertion failure in f is discharged at this call site; otherwise, ϕ'_f is conjoined with the precondition ϕ of the current function.

Next, we compose the partial heap $\mathbb{S}_f$, representing the heap fragment reachable in f after the call, with the existing heap $\mathbb{S}$ before the function call. The *compose_partial_heap* rule used in *compose_heap* instantiates an entry $\pi \mapsto \theta$ in f's summary. Observe that if location π in f's summary instantiates to location π_i in the current function under ϕ_i, existing values of π_i are only preserved under $\neg\phi_i$. Hence, if ϕ_i is *true*, this rule applies a strong update to π_i. On the other hand, if π instantiates to π_i under some entry alias condition, then this rule represents a strong update to π_i only in those contexts where the entry aliasing condition holds.

Example 25 *Consider a call to function f of Example 22:*

$$\texttt{define g}(\texttt{a}_1 : \texttt{ptr(ptr(int))}) = \texttt{f}^3(\texttt{a}_1, \texttt{a}_1)$$

Before the call to f, g's local heap is depicted as:



$$\frac{\begin{array}{l}\mathbb{I}, \rho \vdash \mathit{inst\_loc}(\mathit{lift}^{-1}(t_1)) : \{(\pi_1, \phi_1), \ldots, (\pi_k, \phi_k)\} \\ \mathbb{I}, \rho \vdash \mathit{inst\_loc}(\mathit{lift}^{-1}(t_2)) : \{(\pi'_1, \phi'_1), \ldots, (\pi'_m, \phi'_m)\} \\ \phi = \bigvee_i (k = \pi_i \wedge \phi_i) \quad \phi' = \bigvee_j (k' = \pi'_j \wedge \phi'_j) \;\; (k, k' \textit{ fresh})\end{array}}{\mathbb{I}, \rho \vdash \mathit{inst}_\varphi(t_1 = t_2) : k = k', \phi \wedge \phi'}$$

$$\frac{b \in \{T, F\}}{\mathbb{I}, \rho \vdash \mathit{inst}_\varphi(b) : b, T} \quad \frac{\mathbb{I}, \rho \vdash \mathit{inst}_\varphi(\varphi) : \varphi_1, \phi_2}{\mathbb{I}, \rho \vdash \mathit{inst}_\varphi(\neg\varphi) : \neg\varphi_1, \phi_2}$$

$$\frac{\mathbb{I}, \rho \vdash \mathit{inst}_\varphi(\varphi_1) : \varphi, \phi \quad \mathbb{I}, \rho \vdash \mathit{inst}_\varphi(\varphi_2) : \varphi', \phi'}{\mathbb{I}, \rho \vdash \mathit{inst}_\varphi(\varphi_1 \star \varphi_2) : \varphi \star \varphi', \phi \wedge \phi'} (\star \in \{\wedge, \vee\})$$

$$\frac{\begin{array}{l}\mathbb{I}, \rho \vdash \mathit{inst}_\varphi(\varphi_{may}) : \varphi'_{may}, \phi'_{may} \quad \mathbb{I}, \rho \vdash \mathit{inst}_\varphi(\varphi_{must}) : \varphi'_{must}, \phi'_{must} \\ \varphi''_{may} = \lceil QE(\exists \vec{k}.\ (\varphi'_{may} \wedge \phi'_{may})) \rceil \quad \varphi''_{must} = \lfloor QE(\exists \vec{k}.\ (\varphi'_{must} \wedge \phi'_{must})) \rfloor\end{array}}{\mathbb{I}, \rho \vdash \mathit{inst}_\phi(\langle \varphi_{may}, \varphi_{must} \rangle) : \langle \phi''_{may}, \phi''_{must} \rangle}$$

Figure 4.14: Rules for instantiating constraints

$$\frac{\mathbb{I}, \rho \vdash \mathit{inst\_loc}(\pi_1) = \theta_1, \ldots \mathit{inst\_loc}(\pi_k) = \theta_k}{\mathbb{I}, \rho \vdash \mathit{inst\_theta}(\{(\pi_1, \phi_1), \ldots, (\pi_k, \phi_k)\}) : \bigcup_{1 \le i \le k} (\theta_i \wedge \phi_i)}$$

$$\frac{\begin{array}{c}\mathbb{I}, \rho \vdash \mathit{inst\_loc}(\pi) = \theta_s \\ \mathbb{I}, \rho \vdash \mathit{inst\_theta}(\theta) = \theta_t \\ \mathbb{S}' = \mathbb{S}[\pi_i \leftarrow (\theta_t \wedge \phi_i) \cup (\mathbb{S}(\pi_i) \wedge \neg\phi_i) \mid (\pi_i, \phi_i) \in \theta_s]\end{array}}{\mathbb{S}, \mathbb{I}, \rho \vdash \mathit{compose\_partial\_heap}(\pi, \theta) : \mathbb{S}'}$$

$$\frac{\begin{array}{c}\mathbb{S}_f = [(\pi_1 \mapsto \theta_1), \ldots, (\pi_k \mapsto \theta_k)] \\ \mathbb{S}, \mathbb{I}, \rho \vdash \mathit{compose\_partial\_heap}(\pi_1, \theta_1) : \mathbb{S}_1 \\ \ldots \\ \mathbb{S}_{k-1}, \mathbb{I}, \rho \vdash \mathit{compose\_partial\_heap}(\pi_k, \theta_k) : \mathbb{S}_k\end{array}}{\mathbb{S}, \mathbb{I}, \rho \vdash \mathit{compose\_heap}(\mathbb{S}_f) : \mathbb{S}_k}$$

$$\frac{\begin{array}{l}\mathbb{G}(f) = \langle \phi_f, \mathbb{S}_f \rangle \\ \mathbb{E}, \mathbb{S} \vdash \mathit{map\_args}(v_1, \ldots, v_k) : \mathbb{I} \\ \mathbb{I}, \rho \vdash \mathit{inst}_\phi(\phi_f) : \phi'_f \\ \mathbb{S}, \mathbb{I}, \rho \vdash \mathit{compose\_heap}(\mathbb{S}_f) : \mathbb{S}'\end{array}}{\mathbb{E}, \mathbb{S}, \mathbb{G}, \phi \vdash f^\rho(v_1, \ldots, v_k) : \mathbb{S}', \phi \wedge \phi'_f}$$

Figure 4.15: Summary Instantiation rules

Recall from Example 22 that f's precondition is $\mathrm{drf}(\nu_1) \neq \mathrm{drf}(\nu_2)$*, which instantiates to* $\mathrm{drf}(\nu_1) \neq \mathrm{drf}(\nu_1) \Leftrightarrow \mathrm{false}$ *at this call site, indicating that the assertion is guaranteed to fail. The store in f's summary from Figure 4.7 is instantiated at the call site to:*

$$\begin{tikzcd} *\nu_1 \arrow[r, "T"] & loc^{3,2} \arrow[r, "T"] & 8 \end{tikzcd} \quad (**)$$

Composing initial heap (*) with the instantiated heap (**), we obtain the final heap after the function call:

$$\begin{tikzcd} \nu_1 \arrow[r, "T"] & *\nu_1 \arrow[r, "T"] & loc^{3,2} \arrow[r, "T"] & 8 \end{tikzcd}$$

Observe that the resulting abstract heap is as precise as analyzing the inlined body of f.

Summary Generation and Fixed-point Computation

We now conclude this section by describing function summary generation, given in Figure 4.16. Before analyzing the body of f, the local abstract heap $\mathbb{S}$ is initialized as described in Section 4.3.3. Next, f's body is analyzed using the abstract transformers from Section 4.3.3 and 4.3.3, which yields a store $\mathbb{S}'$ and a precondition ϕ'. According to the last hypothesis in Figure 4.16, the summary $\langle \phi_f, \mathbb{S}_f \rangle$ is sound if $\mathbb{S}_f$ overapproximates $\mathbb{S} \backslash \{\nu_1, \ldots, \nu_k\}$ and ϕ_f implies ϕ'. Here, $\mathbb{S}_1 \sqsubseteq \mathbb{S}_2$ is defined as:

Definition 22 *($\sqsubseteq$) Let $\mathbb{S}'_1 = \mathbb{S}_{1 \mapsto \mathbb{S}_2}$ and $\mathbb{S}'_2 = \mathbb{S}_{2 \mapsto \mathbb{S}_1}$. We say $\mathbb{S}_1 \sqsubseteq \mathbb{S}_2$ if for every $\pi \in$* $\mathrm{dom}(\mathbb{S}'_1)$ *and for every π' such that $(\pi', \langle \varphi^1_{\mathrm{may}}, \varphi^1_{\mathrm{must}} \rangle) \in \mathbb{S}'_1(\pi)$, $(\pi', \langle \varphi^2_{\mathrm{may}}, \varphi^2_{\mathrm{must}} \rangle) \in \mathbb{S}'_2(\pi)$, we have:*

$$\varphi^1_{\mathrm{may}} \Rightarrow \varphi^2_{\mathrm{may}} \wedge \varphi^2_{\mathrm{must}} \Rightarrow \varphi^1_{\mathrm{must}}$$

While the rule in Figure 4.16 verifies that $\langle \phi_f, \mathbb{S}_f \rangle$ is a sound summary, it does not give an algorithmic way of computing it. In the presence of recursion, we perform a least fixed-point computation where all entries in $\mathbb{G}$ are initially $\perp$ (i.e., any location points to any other location under *false*), and a new summary for f is obtained by computing the join of f's new and old summaries:

$$\langle \mathbb{S}_1, \phi_1 \rangle \sqcup \langle \mathbb{S}_2, \phi_2 \rangle = \langle \mathbb{S}_1 \sqcup \mathbb{S}_2, \phi_1 \wedge \phi_2 \rangle$$

$$\frac{\begin{array}{l}\mathbb{A} \vdash \mathit{init\_heap}(a_1, \ldots, a_k) : \mathbb{E}, \mathbb{S} \\ \mathbb{E}, \mathbb{S}, \mathbb{G}, \mathit{true} \vdash S : \phi', \mathbb{S}' \\ \mathbb{G} \vdash f : \langle \phi_f, \mathbb{S}_f \rangle \\ \phi_f \Rightarrow \phi' \quad \mathbb{S}_f \sqsupseteq (\mathbb{S}' \backslash \{\nu_1, \ldots, \nu_k\})\end{array}}{\mathbb{G}, \mathbb{A} \vdash \mathit{define}\ f(a_1 : \tau_1, \ldots, a_k : \tau_k) = S : \langle \phi_f, \mathbb{S}_f \rangle}$$

Figure 4.16: Summary generation rule

This strategy ensures that the analysis is monotonic by construction. Furthermore, since the analysis creates a finite number of abstract locations and the constraints are over a finite vocabulary of predicates, this fixed-point computation is guaranteed to converge. In fact, for an acyclic callgraph, each function is analyzed only once if a reverse topological order is used.

4.4 Computing Alias Partition Sets

In the previous section, we assumed the existence of an alias partition environment $\mathbb{A}$ that is used to query whether aliasing between locations α and α' may affect analysis results. One simple way to compute such an environment is to require that $\alpha' \in \mathbb{A}(\alpha)$ if α and α' have the same type (at least in a type-safe language). Fortunately, it is possible to compute a much more precise alias partition environment because many aliasing relations at a call site of f do not affect the state of the heap after a call to f. The following lemma elucidates when we can safely ignore potential aliasing between two locations in a code fragment S.

Lemma 11 *Let H_1 and H_2 be the canonical heap fragments shown in Figure 4.17, and let S be a program fragment such that:*

- *There is either no store to A and no store to B, or*
- *There is a store to only A that is not followed by a load from B, or*

$$\Pi_{H_2,H_1}(E) = C$$
$$\Pi_{H_2,H_1}(G) = C$$
$$\Pi_{H_2,H_1}(F) = D$$
$$\Pi_{H_2,H_1}(H) = D$$

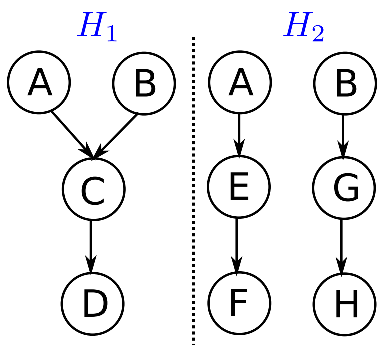


Figure 4.17: Heaps from Lemma 11

- *There are only stores to both A and B, but the store to A must happen after the store to B*

Let $H_1 \vdash S : H_1'$ and $H_2 \vdash S : H_2'$, and let O be a partial order such that $O(B) \prec O(A)$ if there must be a write to A after a write to B in S. Let H_2'' be the graph obtained by replacing G's targets with E's targets in H_2' if $\Pi_{H_2,H_1}(A) = \Pi_{H_2,H_1}(B)$ and $O(B) \prec O(A)$. Then, $H_1' = \Pi_{H_2,H_1}(H_2'')$.

Proof 11 *(sketch) There are three cases: (i) If there is no store to A or B, then in H_2', E and G still point to F and H, both of which are equivalent to D in H_1'. Thus, $H_1' = \Pi_{H_2,H_1}(H_2')$. (ii) There is only a store to A, not followed by a load from B: In H_1', C will point to some set of new locations $T_1, \ldots, T_k$. In H_2', E must also point to $T_1', \ldots, T_k'$ such that $T_i = \Pi_{H_2,H_1}(T_i')$ and G must point to H. First, the result of any load from B (i.e., H) can be correctly renamed to D, as the read happens before the store to A. Second, H_2'' is obtained by removing the edge from G to H and adding edges from G to each T_i'. Thus, $\Pi_{H_2,H_1}(G) = \Pi_{H_2,H_1}(E) = C$ and G and E's targets are renamed to $T_1, \ldots, T_k$. (iii) Similar to (ii).*

This lemma shows the principle that can be used to reduce the number of entries in $\mathbb{A}$: Assuming we can impose an order on the sequence of updates to memory locations and assuming we instantiate summary edges in this order, then the initial heap abstraction only needs to account for aliasing between α_1 and α_2 if there is a store to α_1 followed by a load from α_2, which is necessary because updates through

α_1 to a location may now affect locations that are reachable through α_2. On the other hand, if there is no load after a store and the updates to memory locations can be ordered, it is possible to "fix up" the summary at the call site by respecting the order of updates during instantiation.

To allow such an optimization in the analysis described in Section 4.3, we impose a partial order $\prec$ on points-to relations such that $(\pi_1 \mapsto \theta_1) \prec (\pi_2 \mapsto \theta_2)$ indicates that π_1 must be assigned to θ_1 before π_2 is assigned to θ_2. Then, to respect the order of updates in the callee when instantiating the summary, we ensure that if $\pi_i \mapsto \theta_i \prec \pi_j \mapsto \theta_j$, the *compose_partial_heap* rule is invoked on $\pi_i \mapsto \theta_i$ before $\pi_j \mapsto \theta_j$ in the *compose_heap* rule of Figure 4.15.

Thus, assuming we modify the analysis from Section 4.3 as described above, we can compute a better alias partition environment $\mathbb{A}$ by performing a least fixed-point computation over the current function f. In particular, $\mathbb{A}(\alpha)$ is initialized to $\{\alpha\}$ for each location variable α. Then, if the analysis detects a store to α followed by a load from α' of the same type, then $\alpha' \in \mathbb{A}(\alpha)$ and $\alpha \in \mathbb{A}(\alpha')$. Similarly, if there is a store s_1 to α and a store s_2 to α' (of the same type) such that there is no happens-before relation between s_1 and s_2, then $\alpha' \in \mathbb{A}(\alpha)$ and $\alpha \in \mathbb{A}(\alpha')$.

4.5 Experiments

We have implemented the technique described in this chapter in our Compass program verification framework for analyzing C and C++ applications. Our implementation extends the algorithm described in this chapter in two ways: First, our analysis is fully (i.e., interprocedurally) path-sensitive and uses the algorithm of Chapter 3 for this purpose. Second, our implementation improves over the analysis presented here by employing the technique described in [32], which uses *indexed locations* to reason precisely about contents of arrays and containers. Hence, the algorithm we implemented is significantly more precise than a standard may points-to analysis.

Figure 4.18 summarizes the results of our first experiment, which involves verifying memory safety properties (buffer overruns, null dereferences, casting errors, and access to deleted memory) in four real C and C++ applications ranging from 16,030 to

	LiteSQL	**OpenSSH**	**Inkscape widget lib.**	**Digikam**
Lines	16,030	22,615	37,211	128,318
Strong updates at instantiation				
Running time (1 CPU)	4.5 min	3.9 min	7.2 min	45.1 min
Running time (8 CPUs)	1.6 min	1.8 min	2.3 min	8.7 min
Memory use	430 MB	230 MB	195 MB	400 MB
Error reports	7	6	7	37
False positives	2	1	3	9
Weak updates at instantiation				
Running time (1 CPU)	7.1 min	4.8 min	8.1 min	60.0 min
Running time (8 CPUs)	4 min	3.6 min	2.5 min	10.1 min
Memory use	410 MB	250 MB	200 MB	355 MB
Error reports	312	209	730	1140
False positives	307	204	726	1112

Figure 4.18: Comparison of strong/weak updates at call sites

128,318 lines. The first part of the table, labeled “Strong Updates at Instantiation”, reports the results obtained by using the modular heap analysis described in this chapter. Observe that the proposed technique is both scalable, memory-efficient, and precise. First, the running times on 8 CPU’s range from 1.6 minutes to 8.7 minutes, and increase roughly linearly with the size of the application. Furthermore, observe that the modular analysis takes advantage of multiple CPUs to significantly reduce its wall-clock running time. Second, the maximum memory used by any process does not exceed 430 MB, and, most importantly, the memory usage is not correlated with the application size. Figure 4.19 sheds some light on the scalability of the analysis: This figure plots the maximum call stack depth against summary size, computed as the number of points-to edges weighted according to the size of the edge constraints plus the size of the precondition. In this figure, observe that summary size does not increase with depth of the callstack, confirming our hypothesis that summaries are useful for exploiting information locality and therefore enable analyses to scale.

Figure 4.18 also illustrates that performing strong updates at call sites is crucial for the precision required by verification tools. Observe that the analysis using strong

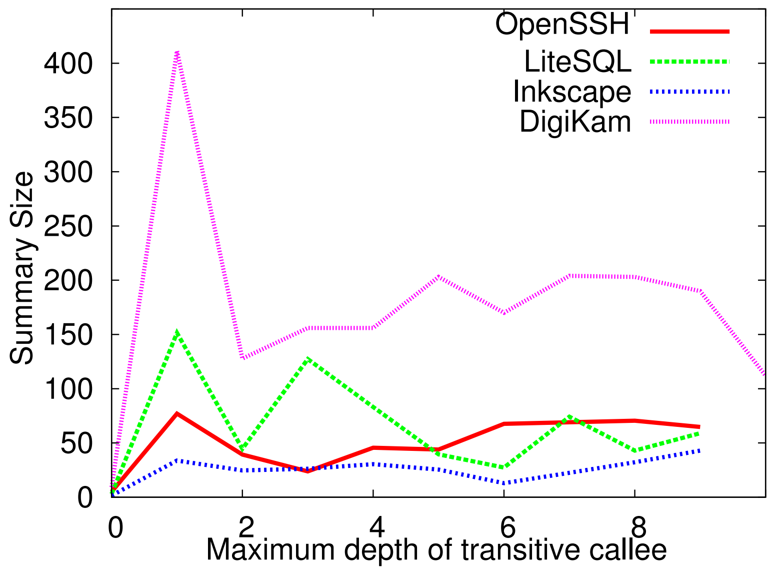


Figure 4.19: Callstack depth vs. summary size

	hostname	**chroot**	**rmdir**	**su**	**mv**
Lines	304	371	483	1047	1151
Modular analysis running time	0.53s	0.75s	1.54s	2.3s	2.55s
Whole program running time	3.1s	6.3s	21.6s	45.9s	30.7s

Figure 4.20: Comparison of modular and whole program analysis

updates at instantiation sites is very precise, reporting only a handful of false positives on all the applications. In contrast, if we use only weak updates when applying summaries, the number of false positives ranges from 200 to 1000, confirming that the application of strong updates interprocedurally is a key requirement for successful verification.

In a second set of experiments on smaller benchmarks, we compare the running times of our verification tool using the modular analysis described here with the running times of the same tool using a whole-program analysis. Figure 4.20 shows a comparison of the analysis running times of the modular and whole program analysis on five Unix Coreutils applications. As shown in this figure, the whole program

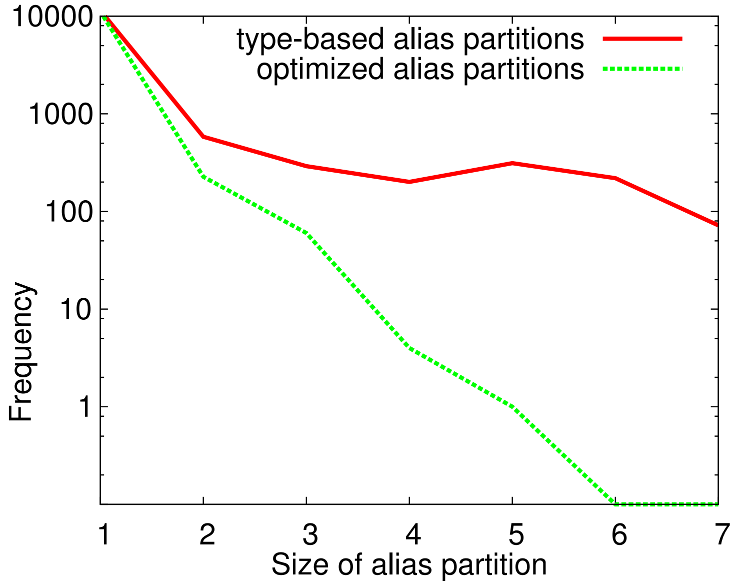


Figure 4.21: Size of alias partition set vs. Frequency

analysis, which did not report any errors, takes ∼50 seconds on a program with only 1000 lines, whereas the modular analysis, which also did not report any errors, analyzes the same program in 2.3 seconds. Furthermore, observe that the running time of the whole program analysis increases much more quickly in the size of the application than that of the modular analysis.

In a final set of experiments, we plot the size of the alias partition set vs. the frequency of this set size for the benchmarks from Figure 4.18. The solid (red) line shows the size of the alias partition sets obtained by assuming $\alpha' \in \mathbb{A}(\alpha)$ if α and α' have compatible types. In contrast, the dashed (green) line shows the size of the alias partition sets obtained as described in Section 4.4. Observe that these optimizations significantly reduce the size of alias partition sets and substantially improve running time. In particular, without these optimizations, the benchmarks take an average of 2.7 times longer.

4.6 Related Work

In this section, we review previous approaches to compositional program analysis, specifically compositional alias analysis, compositional shape analysis and general modular analysis frameworks. Among those approaches, previous work on compositional alias analysis is most related to the work presented in this chapter.

Compositional Alias Analysis Modular alias analysis of a procedure performed by starting with unknown values to all parameters was also explored in [79] and then in Relevant Context Inference (RCI) [24]. The technique presented in [79] computes a new *partial transfer function* as new aliasing patterns are encountered at call sites and requires reanalysis of functions. In contrast, the technique in [24] is purely bottom-up, and uses equality and disequality queries to generate *summary transfer functions*. Our approach is similar to [24] in that we perform a strictly bottom-up analysis where the unknown points-to target of an argument is represented using one location variable and summary facts are predicated upon possible aliasing patterns at function entry. In contrast to our technique, RCI is only able to perform strong updates in very special cases intraprocedurally, and cannot perform strong updates at call sites. In fact, the summary computation described in [24] is only sound under the assumption that no points-to relations are killed by summary application. In contrast, summaries generated by our analysis are used to perform strong updates at call sites, and for the recursion-free fragment of the language from Section 4.2, applying a summary is as precise as analyzing the inlined body of the function.

The compositional pointer analysis algorithms given in [78, 75] assume there is no aliasing on function entry and analyze the function body under this assumption. However, since summaries computed in this way may be unsound, the summary is "corrected" using a fairly involved fixed-point computation at call sites. This approach is also much less precise than our technique because it only performs strong updates in a very limited number of situations.

Compositional Shape Analysis Recently, there has also been interest in compositional shape analysis using separation logic [21, 45]. Both of these works use *bi-abduction* to compute pre- and post-conditions on the shapes of recursive data structures. However, neither of these works guarantee precision. While this chapter does not address computing summaries about shapes of recursive data structures, our technique can handle deep sharing and allows disjunctive facts.

General Modular Analysis Frameworks Theoretical foundations for modular program analysis are explored in [28], [46], and [70]. The work in [82] provides a framework for computing precise and concise summaries for IFDS [73] and IDE [74] dataflow problems. This framework is mainly specialized for typestate properties and relies on global points-to information. While it may be possible to apply this framework to obtain some form of modular heap analysis in principle, it is unclear how to do so, and the authors of [82] list this application as a future research direction.

Chapter 5

Constraint Simplification

Static analysis techniques, such as the ones we have described in previous chapters, build upon SAT and SMT solving by encoding program states as formulas and determining the feasibility of these states by querying satisfiability. Despite tremendous progress in solving SAT and SMT formulas over the last decade [36, 56, 62, 31, 35, 14, 9, 12], the scalability of many software verification techniques relies crucially on controlling the size of the formulas generated by the analysis, because many of the operations performed on these formulas are highly sensitive to formula size. For this reason, much research effort has focused on identifying only those states and predicates relevant to some property of interest. For example, *predicate abstraction*-based approaches using *counter-example guided abstraction refinement* [27, 10, 8] attempt to discover a small set of predicates relevant to verifying a property and only include this small set of predicates in their formulas. Similarly, many path-sensitive static analysis techniques have successfully employed various heuristics to identify which path conditions are likely to be relevant for some property of interest. For example, *property simulation* only tracks those branch conditions for which the property-related behavior differs along the arms of the branch [30]. Other path-sensitive analysis techniques attempt to improve their scalability by either only tracking path conditions intraprocedurally or by heuristically selecting a small set of predicates to track across function boundaries [1, 18].

All of these different techniques share one important underlying assumption that has been validated by a large body of empirical evidence: Many program conditions do not matter for verifying most properties of interest, making it possible to construct much smaller formulas sufficient to prove the property. If this is indeed the case, then one might suspect that even if we construct a formula ϕ characterizing some program property P without being particularly careful about what conditions to track, it should be possible to use ϕ to construct a much smaller, equivalent formula ϕ' for P since many predicates used in ϕ do not affect P's truth value.

In this chapter, we present a systematic and practical approach for simplifying formulas that identifies and removes irrelevant predicates and redundant subexpressions as they are generated by the analysis. In particular, given an input formula ϕ, our technique produces an equivalent formula ϕ' such that no simpler equivalent formula can be obtained by replacing any subset of the *leaves* (i.e., syntactic occurrences of atomic formulas) used in ϕ' by *true* or *false*. We call such a formula ϕ' *simplified*.

Like all the afore-mentioned approaches to program verification, our interest in simplification is motivated by the goal of generating formulas small enough to make software verification scalable. However, we attack the problem from a different angle: Instead of restricting the set of predicates that are allowed to appear in formulas, we continuously simplify the constraints generated by the analysis. This approach has two advantages: First, it does not require heuristics to decide which predicates are relevant, and second, this approach removes all redundant subparts of a formula in addition to filtering out irrelevant predicates.

To be concrete, consider the following code snippet:

```
enum op_type {ADD=0, SUBTRACT=1, MULTIPLY=2, DIV=3};
int perform_op(op_type op, int x, int y) {
   int res;
   if(op == ADD) res = x+y;
   else if(op == SUBTRACT) res = x-y;
   else if(op == MULTIPLY) res = x*y;
   else if(op == DIV) { assert(y!=0); res = x/y; }
   else res = UNDEFINED;
   return res; }
```

The `perform_op` function is a simple evaluation procedure inside a calculator program that performs a specified operation on `x` and `y`. This function aborts if the specified operation is division and the divisor is 0. Assume we want to know the constraint under which the function returns, i.e., does not abort. This constraint is given by the disjunction of the constraints under which each branch of the `if` statement does not abort. The following formula, constructed in a straightforward way from the program, describes this condition:

$$\begin{gathered} op = 0 \vee (op \neq 0 \wedge op = 1) \vee (op \neq 0 \wedge op \neq 1 \wedge op = 2) \vee \\ (op \neq 0 \wedge op \neq 1 \wedge op \neq 2 \wedge op = 3 \wedge y \neq 0) \vee \\ (op \neq 0 \wedge op \neq 1 \wedge op \neq 2 \wedge op \neq 3) \end{gathered}$$

Here, each disjunct is associated with one branch of the `if` statement. In each disjunct, a disequality constraint of the form $op \neq 0, op \neq 1, \ldots$ states that the previous branches were not taken, encoding the semantics of an `else` statement. In the fourth disjunct, the additional constraint $y \neq 0$ encodes that if this branch is taken, `y` cannot be 0 for the function to return.

While this automatically generated constraint faithfully encodes the condition under which the function returns, it is far from concise. In fact, the above constraint is equivalent to the much simpler formula:

$$op \neq 3 \vee y \neq 0$$

This formula is in *simplified form* because it is equivalent to the original formula and replacing any of the remaining leaves by *true* or *false* would not result in an equivalent formula. This simpler constraint expresses exactly what is relevant to the function's return condition and makes no reference to irrelevant predicates, such as $op = 0, op = 1$, and $op = 2$. Although the original formula corresponds to a brute-force enumeration of all paths in this function, its simplified form yields the most concise representation of the function's return condition without requiring specialized techniques for identifying relevant predicates.

To summarize, this chapter makes the following key contributions:

- We present an *on-line constraint simplification* algorithm for improving SMT-based static analysis techniques.
- We define what it means for a formula to be in *simplified form* and detail some important properties of this form.
- We give a practical algorithm for reducing formulas to their simplified form and show how this algorithm naturally integrates into the DPLL($\mathcal{T}$) framework for solving SMT formulas.
- We demonstrate the effectiveness of our on-line simplification algorithm in the context of a program verification framework and show that simplification improves overall performance by orders of magnitude, often allowing analysis runs that did not terminate within the allowed resource limits to complete in just a few seconds.

5.1 Preliminaries

Any quantifier-free formula $\phi_{\mathcal{T}}$ in theory $\mathcal{T}$ is defined by the following grammar:

$$\phi_{\mathcal{T}} := \mathit{true} \mid \mathit{false} \mid A_{\mathcal{T}} \mid \neg A_{\mathcal{T}} \mid \phi'_{\mathcal{T}} \wedge \phi''_{\mathcal{T}} \mid \phi'_{\mathcal{T}} \vee \phi''_{\mathcal{T}}$$

In the above grammar, $A_{\mathcal{T}}$ represents an *atomic formula* in theory $\mathcal{T}$, such as the boolean variable x in propositional logic or the inequality $a + 2b \leq 3$ in linear arithmetic. Observe that the above grammar requires formulas to be in *negation normal form (NNF)* because only atomic formulas may be negated. While the rest of this chapter relies on formulas being in NNF, this restriction is not important since any formula may be converted to NNF using De Morgan's laws in linear time without increasing the size of the formula (see Definition 2).

Definition 1 (Leaf) *We refer to each occurrence of an atomic formula $A_{\mathcal{T}}$ or its negation $\neg A_{\mathcal{T}}$ as a* leaf *of the formula in which it appears.*

It is important to note that different occurrences of the same (potentially negated) atomic formula in $\phi_{\mathcal{T}}$ form distinct leaves. For example, the two occurrences of $f(x) = 1$ in $f(x) = 1 \vee (f(x) = 1 \wedge x + y \leq 1)$ correspond to two distinct leaves. Also, observe that leaves are allowed to be negations. For instance, in the formula $\neg(x = y)$, $(x = y)$ is not a leaf; the only leaf of the formula is $\neg(x = y)$.

In the rest of this chapter, we restrict our focus to quantifier-free formulas in theory $\mathcal{T}$, and we assume there is a decision procedure $D_{\mathcal{T}}$ that can be used to decide the satisfiability of a quantifier-free formula $\phi_{\mathcal{T}}$ in theory $\mathcal{T}$. Where irrelevant, we omit the subscript $\mathcal{T}$ and denote formulas by ϕ.

Definition 2 (Size) *The* size *of a formula ϕ is the number of leaves ϕ contains.*

Definition 3 (Fold) *The* fold *operation removes constant leaves (i.e.,* true*,* false*) from the formula. In particular,* Fold(ϕ) *is a formula ϕ' such that (i) $\phi \Leftrightarrow \phi'$, (ii) ϕ' is just* true *or* false *or ϕ' mentions neither* true *nor* false*.*

It is easy to see that it is possible to construct this fold operation such that it reduces the size of the formula ϕ at least by one if ϕ contains *true* or *false* but ϕ is not initially *true* or *false*.

5.2 Simplified Form

In this section, we first define *redundancy* and describe what it means for a formula to be in *simplified form*. We then highlight some important properties of simplified forms. Notions of redundancy similar to ours have been studied in other contexts, such as in *automatic test pattern generation* and *vacuity detection*; see Section 5.6 for a discussion.

Definition 4 ($\boldsymbol{\phi^+}$(L), $\boldsymbol{\phi^-}$(L)) *Let ϕ be a formula and let L be a leaf of ϕ. $\phi^+(L)$ is obtained by replacing L by* true *and applying the fold operation. Similarly, $\phi^-(L)$ is obtained by replacing L by* false *and folding the resulting formula.*

Example 26 *Consider the formula:*

$$\underbrace{x = y}_{L_0} \wedge (\underbrace{f(x) = 1}_{L_1} \vee (\underbrace{f(y) = 1}_{L_2} \wedge \underbrace{x + y \leq 1}_{L_3}))$$

Here, $\phi^+(L_1)$ is $(x = y)$, and $\phi^-(L_2)$ is $(x = y \wedge f(x) = 1)$.

Observe that for any formula ϕ, $\phi^+(L)$ is an overapproximation of ϕ, i.e., $\phi \Rightarrow \phi^+(L)$, and $\phi^-(L)$ is an underapproximation, i.e., $\phi^-(L) \Rightarrow \phi$. This follows immediately from Definition 4 and the monotonicity of NNF. Also, by construction, the sizes of $\phi^+(L)$ and $\phi^-(L)$ are at least one smaller than the size of ϕ.

Definition 5 (Redundancy) *We say a leaf L is* non-constraining *in formula ϕ if $\phi^+(L) \Rightarrow \phi$ and* non-relaxing *if $\phi \Rightarrow \phi^-(L)$. Leaf L is* redundant *if L is either non-constraining or non-relaxing.*

The following corollary follows immediately from definition:

Corollary 3 *If a leaf L is non-constraining, then $\phi \Leftrightarrow \phi^+(L)$, and if L is non-relaxing, then $\phi \Leftrightarrow \phi^-(L)$.*

Intuitively, if replacing a leaf L by *true* in formula ϕ results in an equivalent formula, then L does not constrain ϕ; hence, we call such a leaf non-constraining. A similar intuition applies for non-relaxing leaves.

Example 27 *Consider the formula from Example 26. In this formula, leaves L_0 and L_1 are not redundant, but L_2 is redundant because it is non-relaxing. Leaf L_3 is both non-constraining and non-relaxing, and thus also redundant.*

Note that if two leaves L_1 and L_2 are redundant in formula ϕ, this does not necessarily mean we can obtain an equivalent formula by replacing both L_1 and L_2 with *true* (if non-constraining) or *false* (if non-relaxing). This is the case because eliminating L_1 may render L_2 non-redundant and vice versa.

Definition 6 (Simplified Form) *We say a formula ϕ is in* simplified form *if no leaf mentioned in ϕ is redundant.*

Lemma 12 *If a formula ϕ is in simplified form, replacing any subset of the leaves used in ϕ by* true *or* false *does not result in an equivalent formula.*

Proof 12 *The proof is by induction. If ϕ contains a single leaf, the property trivially holds. Suppose ϕ is of the form $\phi_1 \vee \phi_2$. Then, if ϕ has a simplification $\phi'_1 \vee \phi'_2$ where both ϕ'_1 and ϕ'_2 are simplified, then either $\phi'_1 \vee \phi_2$ or $\phi_1 \vee \phi'_2$ is also equivalent to ϕ. This is the case because $(\phi \Leftrightarrow \phi'_1 \vee \phi'_2) \wedge (\phi \not\Leftrightarrow \phi'_1 \vee \phi_2) \wedge (\phi \not\Leftrightarrow \phi_1 \vee \phi'_2)$ is unsatisfiable. A similar argument applies if the connective is $\wedge$.*

The following corollary follows directly from Lemma 12:

Corollary 4 *A formula ϕ in simplified form is satisfiable if and only if it is not syntactically* false *and valid if and only if it is syntactically* true*.*

This corollary is important in the context of on-line simplification in program analysis because, if formulas are kept in simplified form, then determining satisfiability and validity becomes just a syntactic check.

Observe that while a formula ϕ in simplified form is guaranteed not to contain redundancies, there may still exist a smaller formula ϕ' equivalent to ϕ. In particular, a non-redundant formula may be made smaller, for example, by factoring common subexpressions. We do not address this orthogonal problem in this chapter, and the algorithm given in Section 5.3 does not change the structure of the formula.

Example 28 *Consider the propositional formula $(a \wedge b) \vee (a \wedge c)$. This formula is in simplified form, but the equivalent formula $a \wedge (b \vee c)$ contains fewer leaves.*

As this example illustrates, it is not possible to determine the equivalence of two formulas by checking whether their simplified forms are syntactically identical. Furthermore, as illustrated by the following example, the simplified form of a formula ϕ is not always guaranteed to be unique.

Example 29 *Consider the formula $x = 1 \vee x = 2 \vee (1 \leq x \wedge x \leq 2)$ in the theory of linear integer arithmetic. The two formulas $x = 1 \vee x = 2$ and $1 \leq x \wedge x \leq 2$ are both simplified forms that can be obtained from the original formula.*

Lemma 13 *If ϕ is a formula in simplified form, then* NNF$(\neg\phi)$ *is also in simplified form, where* NNF *converts the formula to negation normal form.*

Proof 13 *Suppose* NNF$(\neg\phi)$ *was not in simplified form. Then, it would be possible to replace one leaf, say L, by* true *or* false *to obtain a strictly smaller, but equivalent formula. Now consider negating the simplified form of* NNF$(\neg\phi)$ *to obtain ϕ' which is equivalent to ϕ. Note that the $\neg L$ is a leaf in ϕ, but not in ϕ'. Thus, ϕ could not have been in simplified form.*

Hence, if a formula is in simplified form, then its negation does not need to be resimplified, an important property for on-line simplification in program analysis. However, simplified forms are not preserved under conjunction or disjunction.

Lemma 14 *For every formula ϕ, there exists a formula ϕ' in simplified form such that (i) $\phi \Leftrightarrow \phi'$, and (ii)* $\text{size}(\phi') \leq \text{size}(\phi)$.

Proof 14 *Consider computing ϕ' by checking every leaf L of ϕ for redundancy and replacing L by* true *if it is non-constraining and by* false *if it is non-relaxing. If this process is repeated until there are no redundant leaves, the resulting formula is in simplified form and contains at most as many leaves as ϕ.*

The above lemma states that converting a formula to its simplified form never increases the size of the formula. This property is desirable because, unlike other representations like BDDs that attempt to describe the formula compactly, computing a simplified form is guaranteed not to cause a worst-case blow-up. In the experience of the authors, this property is crucial in program verification.

5.3 Algorithm to Compute Simplified Forms

While the proof of Lemma 14 sketches a naive way of computing the simplified form of a formula ϕ, this approach is suboptimal because it requires repeatedly checking the satisfiability of a formula twice as large as ϕ until no more redundant leaves can be identified. In this section, we present a practical algorithm to compute simplified forms. For convenience, we assume formulas are represented as trees; however, the algorithm is easily modified to work on directed acyclic graphs, and in fact, our implementation uses DAGs to represent formulas. A node in the tree represents either an $\wedge$ or $\vee$ connective or a leaf. We assume connectives have at least two children but may have more than two.

5.3.1 Basic Algorithm

Recall that a leaf L is non-constraining if and only if $\phi^+(L) \Rightarrow \phi$ and non-relaxing if and only if $\phi \Rightarrow \phi^-(L)$. Since the size of $\phi^+(L)$ and $\phi^-(L)$ may be only one less than ϕ, checking whether L is non-constraining or non-relaxing using Definition 5 requires checking the validity of formulas twice as large as ϕ.

A key idea underlying our algorithm is that it is possible to check for redundancy of a leaf L by checking the validity of formulas no larger than ϕ. In particular, for each leaf L, our algorithm computes a formula $\alpha(L)$, called the *critical constraint* of L, such that (i) $\alpha(L)$ is no larger than ϕ, (ii) L is non-constraining if and only if $\alpha(L) \Rightarrow L$, and (iii) L is non-relaxing if and only if $\alpha(L) \Rightarrow \neg L$. This allows us to determine whether each leaf is redundant by determining the satisfiability of formulas no larger than the original formula ϕ.

Definition 7 (Critical constraint)

- *Let R be the root node of the tree. Then,* $\alpha(R) = \text{true}$.
- *Let N be any node other than the root node. Let P denote the parent of N in the tree, and let $S(N)$ denote the set of siblings of N. Let $\star$ denote $\neg$ if P is an $\vee$ connective, and nothing if P is an $\wedge$ connective. Then,*

$$\alpha(N) = \alpha(P) \wedge \bigwedge_{S_i \in S(N)} \star S_i$$

Intuitively, the critical constraint of a leaf L describes the condition under which L will be relevant for either permitting or disallowing a particular model of ϕ. Clearly, if the assignment to L is to determine whether ϕ is *true* or *false* for a given interpretation,

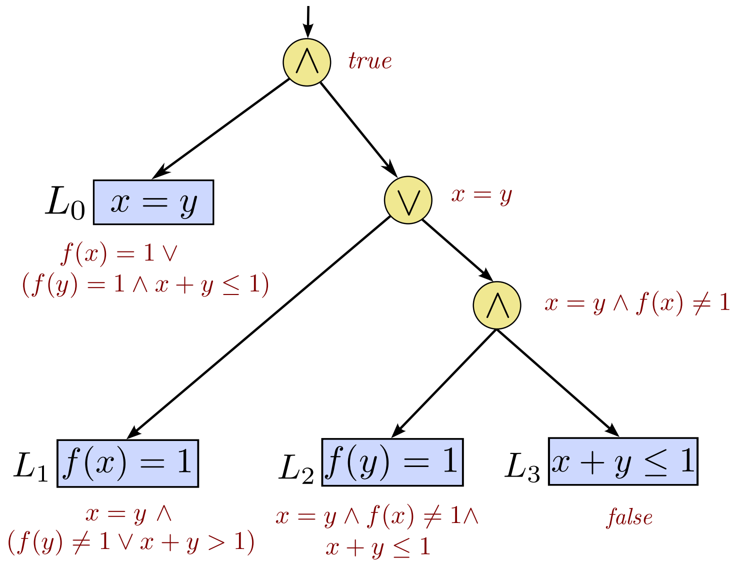


Figure 5.1: The representation of the formula from Example 26. The critical constraint at each node is shown in red. Observe that the critical constraint for L_3 is *false*, making L_3 both non-constraining and non-relaxing. The critical constraint of L_2 implies its negation; hence, L_2 is non-relaxing.

then all the children of an $\wedge$ connective must be true if this $\wedge$ node is an ancestor of L; otherwise ϕ is already false regardless of the assignment to L. Also, observe that L is not relevant in permitting or disallowing a model of ϕ if some other path not involving L is satisfied because ϕ will already be true regardless of the truth value of L. Hence, the critical constraint includes the negation of the siblings at an $\vee$ connective while it includes the siblings themselves at an $\wedge$ node. The critical constraint can be viewed as a *context* in the general framework of *contextual rewriting* [61, 3]; see Section 5.6 for a discussion.

Example 30 *Figure 5.1 shows the representation of the formula from Example 26 along with the critical constraints of each node.*

Lemma 15 *A leaf L is non-constraining if and only if $\alpha(L) \Rightarrow L$.*

Proof 15 *(Sketch) Suppose $\alpha(L) \Rightarrow L$, but L is constraining, i.e., the formula $\gamma = (\phi^+(L) \wedge \neg\phi)$ is satisfiable. Then, there must exist some model M of γ that satisfies*

$\phi^+(L)$ but not ϕ. For M to be a model of $\phi^+(L)$ but not ϕ, it must (i) assign all the children of any $\wedge$ node that is an ancestor of L to true, *(ii) it must assign L to* false, *and (iii) it must assign any other children of an $\vee$ node that is an ancestor of L to* false. *By (i) and (iii), such a model must also satisfy $\alpha(L)$. Since $\alpha(L) \Rightarrow L$, M must also satisfy L, contradicting (ii). The other direction is analogous.*

Lemma 16 *A leaf L is non-relaxing if and only if $\alpha(L) \Rightarrow \neg L$.*

Proof 16 *Similar to the proof of Lemma 15.*

We now formulate a simple recursive algorithm, presented in Figure 5.2, to reduce a formula ϕ to its simplified form. In this algorithm, N is a node representing the current subpart of the formula, and α denotes the critical constraint associated with N. If C is some ordered set, we use the notation $C_{<i}$ and $C_{>i}$ to denote the set of elements before and after index i in C respectively. Finally, we use the notation $\star$ as in Definition 7 to denote $\neg$ if the current node is an $\vee$ connective and nothing otherwise.

Observe that, in the algorithm of Figure 5.2, the critical constraint of each child c_i of a connective node is computed by using the new siblings c'_k that have been simplified. This is crucial for the correctness of the algorithm because, as pointed out in Section 5.2, if two leaves L_1 and L_2 are both initially redundant, it does not mean L_2 stays redundant after eliminating L_1 and vice versa. Using the simplified siblings in computing the critical constraint of c_i has the same effect as rechecking whether c_i remains redundant after simplifying sibling c_k.

Another important feature of the algorithm is that, at connective nodes, each child is simplified as long as any of their siblings change, i.e., the recursive invocation returns a new sibling not identical to the old one. The following example illustrates why this is necessary.

Example 31 *Consider the following formula:* $\underbrace{x \neq 1}_{L_1} \wedge \underbrace{(\underbrace{x \leq 0}_{L_2} \vee \underbrace{x > 2}_{L_3} \vee \underbrace{x = 1}_{L_4})}_{N}$

simplify(N, α)

- If N is a leaf:
  - If $\alpha \Rightarrow N$ return *true*
  - If $\alpha \Rightarrow \neg N$ return *false*
  - Otherwise return N
- If N is a connective, let C denote the ordered set of children of N, and let C' denote the new set of children of N .
  - For each $c_i \in C$:

$$\begin{array}{lcl} \alpha_i & = & \alpha \ \wedge \ (\bigwedge_{c_j \in C_{>i}} \star c_j) \ \wedge (\bigwedge_{c'_k \in C'_{<i}} \star c'_k) \\ c'_i & = & \text{simplify}(c_i, \alpha_i) \\ C' & = & C' \cup c'_i \end{array}$$

  - Repeat the previous step until $\forall i. c_i = c'_i$
  - If N is an $\wedge$ connective, return $\bigwedge_{c'_i \in C'} c'_i$
  - If N is an $\vee$ connective, return $\bigvee_{c'_i \in C'} c'_i$

Figure 5.2: The basic algorithm to reduce a formula N to its simplified form

The simplified form of this formula is $x \leq 0 \vee x > 2$. Assuming we process child L_1 before N in the outer $\wedge$ connective, the critical constraint for L_1 is computed as $x \leq 0 \vee x > 2 \vee x = 1$, which implies neither L_1 nor $\neg L_1$. If we would not resimplify L_1 after simplifying N, the algorithm would (incorrectly) yield $x \neq 1 \wedge (x \leq 0 \vee x > 2)$ as the simplified form of the original formula. However, by resimplifying L_1 after obtaining a simplified $N' = (x \leq 0 \vee x > 2)$, we can now simplify the formula further because the new critical constraint of L_1, $(x \leq 0 \vee x > 2)$, implies $x \neq 1$.

Lemma 17 *The number of validity queries made in the algorithm of Figure 5.2 is bound by $2n^2$ where n denotes the number of leaves in the initial formula.*

Proof 17 *First, observe that if any call to simplify yields a formula different from the input, the size of this formula must be at least one less than the original formula* *(see Lemma 14). Furthermore, the number of validity queries made in formula of size k without any simplifications is $2k$. Hence, the total number of validity queries is bound by $2n + 2(n-1) + \ldots + 2$ which is bound by $2n^2$.*

5.3.2 Making Simplification Practical

In the previous section, we showed that reducing a formula to its simplified form may require making a quadratic number of validity queries. However, these queries are not independent of one another in two important ways: First, all the formulas that correspond to validity queries share exactly the same set of leaves. Second, the simplification algorithm given in Figure 5.2 has a push-and-pop structure, which makes it possible to incrementalize queries. In the rest of this section, we discuss how we can make use of these observations to substantially reduce the cost of simplification in practice.

The first observation that all formulas whose satisfiability is queried during the algorithm share the same set of leaves has a fundamental importance when simplifying SMT formulas. Most modern SMT solvers use the DPLL($\mathcal{T}$) framework to solve formulas [68]. In the most basic version of this framework, leaves in a formula are treated as boolean variables, and this boolean overapproximation is then solved by a SAT solver. If the SAT solver generates a satisfying assignment that is not a valid assignment when theory-specific information is accounted for, the theory solver then produces (an ideally minimal) conflict clause that is conjoined with the boolean overapproximation to prevent the SAT solver from generating at least this assignment in the future. Since the formulas solved by the SMT solver during the algorithm presented in Figure 5.2 share the same set of leaves, theory-specific conflict clauses can be gainfully reused. In practice, this means that after a small number of conflict clauses are learned, the problem of checking the validity of an SMT formula quickly converges to checking the satisfiability of a boolean formula.

The second important observation is that the construction of the critical constraint follows a push-pop stack structure. This is the case because the critical constraint from the parent node is reused, and additional constraints are pushed on the stack (i.e., added to the critical constraint) before the recursive call and (conceptually) popped from the stack after the recursive invocation. This stylized structure is important for making the algorithm practical because almost all modern SAT and SMT solvers support pushing and popping constraints to incrementalize solving. In addition, other tasks that often add overhead, such as CNF construction using Tseitin's encoding for the SAT solver, can also be incrementalized rather than done from scratch. In Section 5.5, we show the expected overhead of simplifying over solving grows sublinearly in the size of the formula in practice if the optimizations described in this section are used.

5.4 Integration with Program Analysis

We implemented the proposed algorithm in the Mistral constraint solver [34]. To tightly integrate simplification into a program analysis system, we designed the interface of Mistral such that instead of giving a "yes/no" answer to satisfiability and validity queries, it yields a formula ϕ' in simplified form. Recall that ϕ is satisfiable (valid) if and only if ϕ' is not syntactically *false* (*true*); hence, in addition to obtaining a simplified formula, the program analysis system can check whether the formula is satisfiable by syntactically checking if ϕ' is not *false*. After a satisfiability query is made, we then replace all instances of ϕ with ϕ' such that future formulas that would be constructed by using ϕ are instead constructed using ϕ'. This functionality is implemented efficiently through a shared constraint representation. Hence, Mistral's interface is designed to be useful for program analysis systems that incrementally construct formulas from existing formulas and make many intermediary satisfiability or validity queries. Examples of such systems include, but are not limited to, [1, 32, 10, 8, 30, 5].

5.5 Experimental Results

In this section, we report on our experience using on-line simplification in the context of program analysis. Since the premise of this work is that simplification is useful

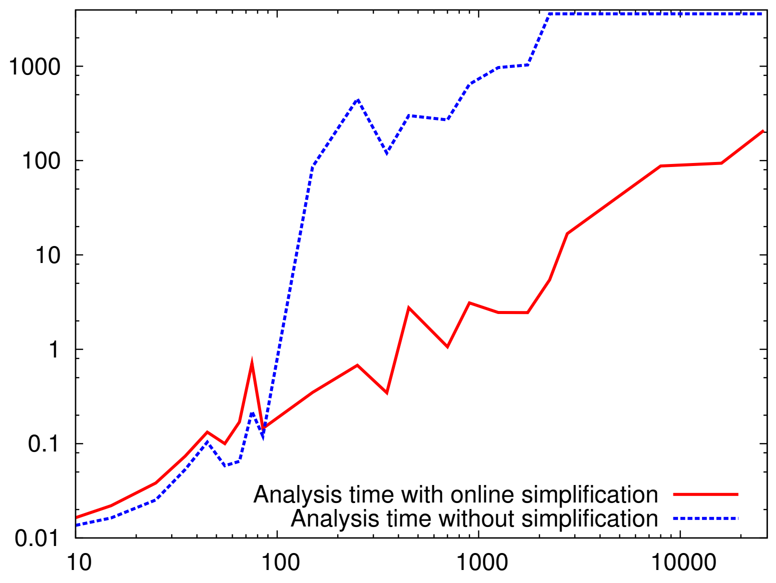


Figure 5.3: Running times with and without simplification

only if applied continuously during the analysis, we do not evaluate the proposed algorithm on solving off-line benchmarks such as the SMT-LIB. In particular, the proposed technique is not meant as a preprocessing step before solving and is not expected to improve solving time on individual constraints.

5.5.1 Impact of On-line Simplification on Analysis Scalability

In our first experiment, we integrate Mistral into the Compass program verification system. Compass [32] is a path- and context-sensitive program analysis system for analyzing C programs, integrating reasoning about both arrays and contents of the heap. Compass checks memory safety properties, such as buffer overruns, null dereferences, casting errors, and uninitialized memory; it can also check user-provided assertions. Compass generates constraints in the combined theory of uninterpreted functions and linear integer arithmetic, and as typical of many program analysis systems [40, 1, 32, 5], constraints generated by Compass become highly redundant over time, as new constraints are obtained by combining existing constraints. Most importantly, unlike other systems that employ various (usually incomplete) heuristics to control formula size, Compass tracks program conditions precisely without identifying a relevant set of predicates to track. Hence, this experiment is used to illustrate that a program analysis system can be made scalable through on-line simplification instead of using specialized heuristics to control formula size.

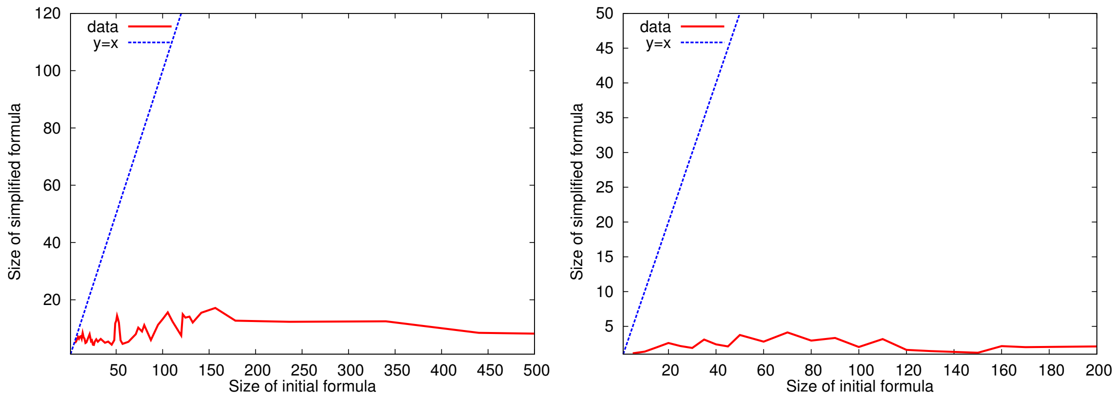
In this experiment, we run Compass on 811 program analysis benchmarks, totalling over 173,000 lines of code, ranging from small programs with 20 lines to real-world applications, such as OpenSSH, with over 26,000 lines. For each benchmark, we fix a time-out of 3600 seconds and a maximum memory of 4 GB. Any run exceeding either limit was aborted and assumed to take 3600 seconds.

Figure 5.3 compares Compass's running times on these benchmarks with and without on-line simplification. The x-axis shows the number of lines of code for various benchmarks and the y-axis shows the running time in seconds. Observe that both axes are log scale. The blue (dotted) line shows the performance of Compass without on-line simplification while the red (solid) line shows the performance of Compass using the simplification algorithm presented in this chapter and using the improvements from Section 5.3.2. In the setting that does not use on-line simplification, Mistral returns the formula unchanged if it is satisfiable and *false* otherwise. As this figure shows, Compass performs dramatically better with on-line simplification on any benchmark exceeding 100 lines. For example, on benchmarks with an average size of 1000 lines, Compass performs about two orders of magnitude better with on-line simplification, and can analyze programs of this size in just a few seconds. Furthermore, using on-line simplification, Compass can analyze benchmarks with a few ten thousand lines of code, such as OpenSSH, in the order of just a few minutes without employing any heuristics to identify relevant conditions.

5.5.2 Redundancy in Program Analysis Constraints

This dramatic impact of simplification on scalability is best understood by considering how redundant formulas become when on-line simplification is disabled when analyzing the same set of 811 program analysis benchmarks. Figure 5.4(a) plots the size of the initial formula vs. the size of the simplified formula when formulas generated by Compass are not continuously simplified. The $x = y$ line is plotted as a comparison to show the worst-case when the simplified formula is no smaller than the original formula. As this figure shows, while formula sizes grow very quickly without on-line simplification, these formulas are very redundant, and much smaller formulas are obtained by simplifying them. We would like to point out that the redundancies present in these formulas cannot be detected through simple syntactic checks because Mistral still performs extensive syntactic simplifications, such as detecting duplicates, syntactic contradictions and tautologies, and folding constants.

To demonstrate that Compass is not the only program analysis system that generates redundant constraints, we also plot in Figure 5.4(b) the original formula size vs. simplified formula size on constraints obtained on the same benchmarks by the Saturn program analysis system [1]. First, observe that the constraints generated by Saturn are also extremely redundant. In fact, their average size after simplification is 1.93 whereas the average size before simplification is 73. Second, observe that the average size of simplified constraints obtained from Saturn is smaller than the average simplified formula size obtained from Compass. This difference is explained by two factors: (i) Saturn is significantly less precise than Compass, and (ii) it adopts heuristics to control formula size.



(a) Size of initial formula vs. size of simplified formula in Compass without simplification

(b) Size of initial formula vs. size of simplified formula in Saturn

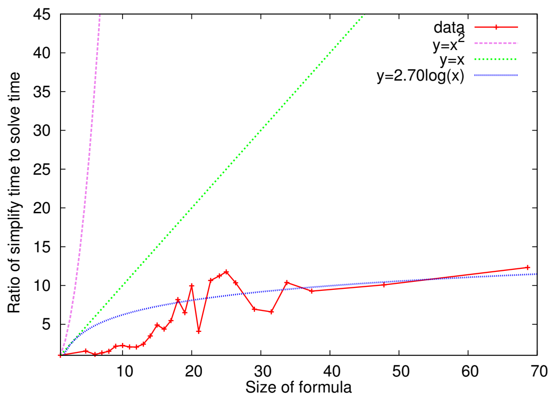
Figure 5.4: Reduction in the Size of Formulas

The reader may not find it surprising that the redundant formulas generated by Compass can be dramatically simplified. That is, of course, precisely the point. Compass gains both better precision and simpler engineering from constructing straightforward formulas and then simplifying them because it does not need to heuristically decide in advance which predicates are important. But these experiments also show that the formulas generated by Compass are not unusually redundant to begin with: As the Saturn experiment shows, because analysis systems build formulas compositionally guided by the structure of the program, even highly-engineered systems like Saturn, designed without the assumption of pervasive simplification, can construct very redundant formulas.

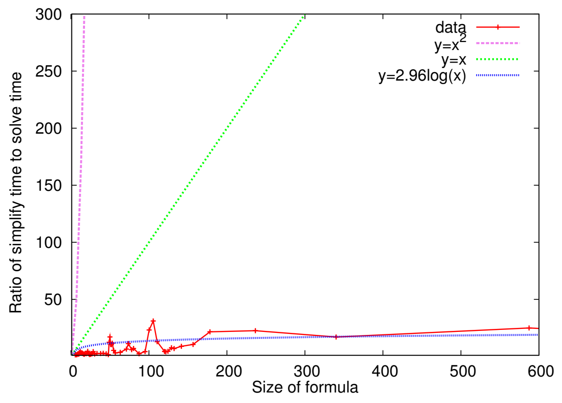
5.5.3 Complexity of Simplification in Practice

In another set of experiments, we evaluate the performance of our simplification algorithm on over 93,000 formulas obtained from our 811 program analysis benchmarks. Recall from Lemma 17 that simplification may require a quadratic number of validity checks. Since the size of the formulas whose validity is checked by the algorithm is at most as large as the original formula, the ratio of simplifying to solving could, in the worst case, be quadratic in the size of the original formula. Fortunately, with the improvements discussed in Section 5.3.2, we show empirically that simplification adds sub-linear overhead over solving in practice.

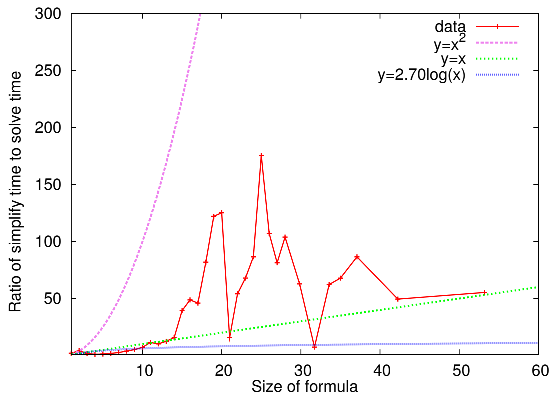
Figure 5.5 shows a detailed evaluation of the performance of the simplification algorithm. In all of these graphs, we plot the ratio of simplifying time to solving time vs. size of the constraints. In graphs 5.5a and 5.5c, the constraints we simplify are obtained from analysis runs where on-line simplification is enabled. For the data in graphs 5.5b and 5.5d, we disable on-line simplification during the analysis, allowing the constraints generated by the analysis to become much larger. We then collect all of these constraints and run the simplification algorithm on these much larger constraints in order to demonstrate that the simplification algorithm also performs well on larger constraints with several hundred leaves. In all of these graphs, the red (solid) line marks data points, the blue (lower dotted) line marks the function best



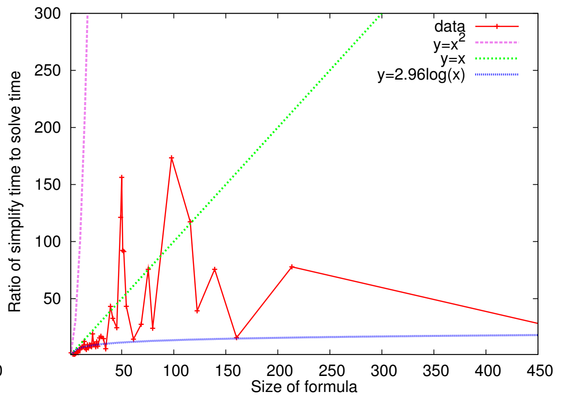
(a)



(b)



(c)



(d)

Figure 5.5: Complexity of Simplification in Practice

fitting the data, the green (middle dotted) line marks $y = x$, and the pink (upper dotted) line marks $y = x^2$. The top two graphs are obtained from runs that employ the improvements described in Section 5.3.2 whereas the two bottom graphs are obtained from runs that do not. Observe that in graphs 5.5a and 5.5b, the average ratio of simplification to solve time seems to grow sublinearly in formula size. In fact, from among the family of formulas $y = cx^2$, $y = cx$, and $y = c \cdot log(x)$, the data in figures 4a and 4b are best approximated by $y = 2.70 \cdot log(x)$ and $y = 2.96 \cdot log(x)$ with asymptotic standard errors 1.98% and 2.42% respectively. On the other hand, runs that do not exploit the dependence between different implication queries exhibit much worse performance, often exceeding the $y = x$ line. These experiments show the importance of exploiting the interdependence between different implication queries and validate our hypothesis that simplifying SMT formulas converges quickly to simplifying SAT formulas when queries are incrementalized. These experiments also show that the overhead of simplifying vs. solving can be made manageable since the ratio of simplifying to solving seems to grow very slowly in the size of the formula.

5.6 Related Work

Finding simpler representations of boolean circuits is a well-studied problem in logic synthesis and automatic test pattern generation (ATPG) [64, 63, 56]. Our definition of redundancy is reminiscent of the concept of *undetectable faults* in circuits, where pulling an input to 0 (false) or 1 (true) is used to identify redundant circuitry. However, in contrast to the definition of size considered in this chapter, ATPG and logic synthesis techniques are concerned with minimizing DAG size, representing the size of the circuit implementing a formula. As a result, the notion of redundancy considered in this chapter is different from the notion of redundancy addressed by these techniques. In particular, in our setting, one subpart of the formula may be redundant while another syntactically identical subpart may not. In this chapter, we consider different definitions of size and redundancy because except for a few operations like substitution, most operations performed on constraints in a program analysis system are sensitive to the "tree size" of the formula, although these formulas are represented as DAGs internally. Therefore, formulas we consider do not exhibit reconvergent fanout and every leaf has exactly one path from the root of the formula. This observation makes it possible to formulate an algorithm based on critical constraints for simplifying formulas in an arbitrary theory. Furthermore, we apply this simplification technique to on-line constraint simplification in program analysis.

The algorithm we present for converting formulas to simplified form can be understood as an instance of a *contextual rewrite system* [61, 3]. In contextual rewriting systems, if a precondition, called a *context*, is satisfied, a rewrite rule may be applied. In our algorithm, the critical constraint can be seen as a context that triggers a rewrite rule $L \rightarrow$ *true* if L is implied by the critical constraint α, and $L \rightarrow$ *false* if α implies $\neg L$. While contextual rewriting systems have been used for simplifying constraints within the solver [3], our goal is to generate an *equivalent* (rather than equisatisfiable) formula that is in simplified form. Furthermore, we propose simplification as an alternative to heuristic-based predicate selection techniques used for improving scalability of program analysis systems.

Finding redundancies in formulas has also been studied in the form of *vacuity detection* in temporal logic formulas [57, 4]. Here, the goal is to identify vacuously valid subparts of formulas, indicating, for example, a specification error. In contrast, our focus is giving a practical algorithm for on-line simplification of program analysis constraints.

The problem of representing formulas compactly has received attention from many different angles. For example, BDDs attempt to represent propositional formulas concisely, but they suffer from the variable ordering problem and are prone to a worst-case exponential blow-up [15]. BDDs have also been extended to other theories, such as linear arithmetic [16, 26, 25]. In contrast to these approaches, a formula in simplified form is never larger than the original formula. Loveland and Shostak address the problem of finding a minimal representation of formulas in normal form [60]; in contrast, our approach does not require formulas to be converted to DNF or CNF.

Various rewrite-based simplification rules have also been successfully applied as a preprocessing step for solving, usually for bit-vector arithmetic [44, 53]. These rewrite rules are syntactic and theory-specific; furthermore, they typically yield equisatisfiable rather than equivalent formulas and give no goodness guarantees. In contrast, the technique described in this chapter is not meant as a preprocessing step for solving and guarantees non-redundancy.

The importance of on-line simplification of program analysis constraints has been studied previously in the very different setting of set constraints [40]. Simplification based on syntactic rewrite-rules has also been shown to improve the performance of a program analysis system significantly in [23].

Finding redundancies in constraints has also been used for optimization of code in the context of constraint logic programming (CLP) [55]. In this setting, constraint simplification is used for improving the running time of constraint logic programs; however, the simplification techniques considered there do not work on arbitrary SMT formulas.

Chapter 6

Conclusion

In this thesis, we have described novel static analysis techniques that make it practical to automatically check low-level safety properties of programs. The analyses we have described are precise enough to report an acceptable number of false alarms and scale to real-world applications. The key ingredients that make our analyses successful are a constraint-based representation of program states, modular algorithms for analyzing the whole program, and continuous simplification of constraints. We believe that these ingredients are crucial for the practicality of our proposed static analysis algorithms, and we believe they can be profitably incorporated into other approaches for static program analysis.

While we have demonstrated that the analyses presented in this thesis are successful for uncovering low-level memory safety errors in real-world applications, an interesting direction for future work is applying the proposed static analysis techniques for automatic detection of violations of high-level specifications, such as security properties. Other interesting applications for the static analysis algorithms described in this thesis include compiler optimizations and program synthesis. More specifically, we believe that the proposed static analysis algorithms are useful for performing much more aggressive compiler optimizations than are performed by current compilers, as they allow practical path-sensitive reasoning and precise, yet scalable scalable alias set computation. We also believe that the proposed static analysis techniques are applicable in the context of program synthesis, where the goal is to generate low-level code from a high-level specification. In particular, an interesting future direction is to employ the techniques described in this thesis for targeting implementations of existing algorithms to new and emerging architectures, such as GPUs.

Bibliography

[1] A. Aiken, S. Bugrara, I. Dillig, T. Dillig, B. Hackett, and P. Hawkins. An overview of the SATURN project. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on program analysis for software tools and engineering*, pages 43–48. ACM, 2007.

[2] A. Aiken, E. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 163–173, 1994.

[3] A. Armando and S. Ranise. Constraint contextual rewriting. *Journal of Symbolic Computation*, 36(1):193–216, 2003.

[4] R. Armoni, L. Fix, A. Flaisher, O. Grumberg, N. Piterman, A. Tiemeyer, and M. Vardi. Enhanced vacuity detection in linear temporal logic. In *Financial Cryptography*, pages 368–380. Springer, 2003.

[5] D. Babic and A.J. Hu. Calysto. In *ICSE'08. ACM/IEEE 30th International Conference on Software Engineering, 2008.*, pages 211–220. IEEE, 2008.

[6] 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.

[7] 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.

[8] T. Ball and S. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 1–3, January 2002.

[9] Clark Barrett and Cesare Tinelli. CVC3. In *Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, July 2007. Berlin, Germany.

[10] D. Beyer, T. Henzinger, R. Jhala, and R. Majumdar. Checking memory safety with Blast. In *Proceedings of the Conf. on Fundamental Approaches to Software Engineering*, pages 2–18, 2005.

[11] R. Bloem, I.H. Moon, K. Ravi, and F. Somenzi. Approximations for fixpoint computations in symbolic model checking. In *Proceedings World Multiconference on Systemics, Cybernetics and Informatics*, volume 8, pages 701–706. Citeseer, 2000.

[12] M. Bofill, R. Nieuwenhuis, A. Oliveras, E. Rodrıguez-Carbonell, and A. Rubio. The Barcelogic SMT Solver. In *Computer Aided Verification*, pages 294–298. Springer-Verlag.

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

[14] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, and Roberto Sebastiani. The MathSAT 4 SMT Solver. In *Computer Aided Verification*, pages 299–303. Springer-Verlag, 2008.

[15] R.E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys (CSUR)*, 24(3):293–318, 1992.

[16] R.E. Bryant and Y.A. Chen. Verification of arithmetic functions with BMDs, 1994.

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

[18] S. Bugrara and A. Aiken. Verifying the safety of user pointer dereferences. In *IEEE Symposium on Security and Privacy, 2008. SP 2008*, pages 325–338, 2008.

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

[20] W. Bush, J. Pincus, and D. Sielaff. A static analyzer for finding dynamic programming errors. *Software Practice and Experience*, 30(7):775–802, 2000.

[21] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. *Proceedings of Principles of Programming Languages*, pages 289–300, 2009.

[22] R. Cartwright and M. Fagan. Soft typing. In *Proceedings of Programming Language Design and Implementation*, pages 278–292, 1991.

[23] Satish Chandra, Stephen J. Fink, and Manu Sridharan. Snugglebug: a powerful approach to weakest preconditions. *SIGPLAN Not.*, 44(6):363–374, 2009.

[24] R. Chatterjee, B.G. Ryder, and W.A. Landi. Relevant context inference. In *Proceedings of Principles of Programming Languages*, pages 133–146. ACM, 1999.

[25] K.C.K. Cheng and R.H.C. Yap. Constrained decision diagrams. In *Proceedings of the National Conference on Artificial Intelligence*, volume 20, page 366, 2005.

[26] E. Clarke, M. Fujita, and X. Zhao. Hybrid decision diagrams overcoming the limitations of MTBDDs and BMDs. In *Proceedings of the 1995 IEEE/ACM international conference on Computer-aided design*, 1995.

[27] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM (JACM)*, 50(5):752–794, 2003.

[28] Patrick Cousot and Radhia Cousot. Modular static program analysis. In *Compiler Construction*, pages 159–178, 2002.

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

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

[31] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[32] I. Dillig, T. Dillig, and A. Aiken. Fluid Updates: Beyond Strong vs. Weak Updates. *Proceedings of European Symposium on Programming Languages 2010*, pages 246–266.

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

[34] I. Dillig, T. Dillig, and A. Aiken. Cuts from proofs: A complete and practical technique for solving linear inequalities over integers. In *Computer Aided Verification*. Springer, 2009.

[35] Bruno Dutertre and Leonardo De Moura. The Yices SMT Solver. Technical report, SRI, 2006.

[36] N. Een and N. Sorensson. MiniSat: A SAT solver with conflict-clause minimization. *8th SAT Conference*, 2005.

[37] D. Engler, D. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. *Operating Systems Review*, 35(5):57–72, 2001.

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

[39] D. Evans. Static detection of dynamic memory errors. In *Proceedings of the Conf. on Programming Language Design and Implementation*, pages 44–53, 1996.

[40] M. Faehndrich, J.S. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *In proceedings of Conference on Programming Languages Design and Implementation*, page 96. ACM, 1998.

[41] M. Faehndrich and K. Rustan M. Leino. Declaring and checking non-null types in an object-oriented language. In *Proceedings of the Conf. on Object-Oriented Programing, Systems, Languages and Applications*, pages 302–312, 2003.

[42] C. Flanagan, R. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the Conf. on Programming Language Design and Implementation*, pages 234–245, 2002.

[43] J. Foster, M. Faehndrich, and A. Aiken. A theory of type qualifiers. In *Proceedings of the Conf. on Programming Language Design and Implementation*, pages 192–203, 1999.

[44] V. Ganesh and D.L. Dill. A decision procedure for bit-vectors and arrays. *Lecture Notes in Computer Science*, 4590:519, 2007.

[45] B. Gulavani, S. Chakraborty, G. Ramalingam, and A. Nori. Bottom-up shape analysis. *Static Analysis Symposium*, pages 188–204, 2009.

[46] S. Gulwani and A. Tiwari. Computing procedure summaries for interprocedural analysis. *Proceedings of European Symposium on Programming Languages*, pages 253–267, 2007.

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

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

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

[50] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, December 2004.

[51] D. Hovemeyer, J. Spacco, and W. Pugh. Evaluating and tuning a static analysis to find null pointer bugs. In *Proceedings of the Workshop on Program Analysis for Software Tools and Engineering*, pages 13–19, 2005.

[52] 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.

[53] S. Jha, R. Limaye, and S.A. Seshia. Beaver: Engineering an Efficient SMT Solver for Bit-Vector Arithmetic. In *In CAV Lecture Notes in Comp. Sc.* Springer, 2009.

[54] R. Jhala and K. McMillan. Interpolant-based transition relation approximation. In *Proceedings of the International Conf. on Computer Aided Verification*, pages 39–51, 2005.

[55] A.D. Kelly, A.M.K. Marriott, P.J. Stuckey, and R.H.C. Yap. Effectiveness of Optimizing Compilation for CLP (R). In *Proceedings of the 1996 Joint International Conference and Symposium on Logic Programming*, page 37. The MIT Press, 1996.

[56] J. Kim, J.P.M. Silva, H. Savoj, and K.A. Sakallah. RID-GRASP: Redundancy identification and removal using GRASP. In *International Workshop on Logic Synthesis*, 1997.

[57] O. Kupferman and M.Y. Vardi. Vacuity detection in temporal model checking. *International Journal on Software Tools for Technology Transfer*, 4(2):224–233, 2003.

[58] W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural aliasing. *SIGPLAN Not.*, 27(7):235–248, 1992.

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

[60] D.W. Loveland and R.E. Shostak. Simplifying interpreted formulas. In *Proceedings 5th Conf. on Automated Deduction (CADE)*, volume 87, pages 97–109. Springer, 1987.

[61] S. Lucas. Fundamentals of Contex-Sensitive Rewriting. *Lecture Notes in Computer Science*, pages 405–412, 1995.

[62] S. Malik, Y. Zhao, C.F. Madigan, L. Zhang, and M.W. Moskewicz. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM, 2001.

[63] A. Mishchenko, R. Brayton, J.H.R. Jiang, and S. Jang. SAT-based logic optimization and resynthesis. *Proceedings IWLS'07*, pages 358–364.

[64] A. Mishchenko, S. Chatterjee, and R. Brayton. DAG-aware AIG rewriting: A fresh look at combinational logic synthesis. In *Proceedings of the 43rd annual Design Automation Conference*, pages 532–535, 2006.

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

[66] M. Naik and J. Palsberg. A type system equivalent to a model checker. In *Proceedings of the European Symposium on Programming*, pages 374–388, 2005.

[67] G. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 128–139, 2002.

[68] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL (T). *Journal of the ACM (JACM)*, 53(6):977, 2006.

[69] F. Pessaux and X. Leroy. Type-based analysis of uncaught exceptions. In *Proceedings of Principles of Programming Languages*, pages 276–290, 1999.

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

[71] 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.

[72] T. W. Reps, S. Sagiv, and R. Wilhelm. Static program analysis via 3-valued logic. In *Proceedings of Conference on Computer Aided Verification*, volume 3114, pages 15–30. Springer, 2004.

[73] Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of Principles of Programming Languages*, pages 49–61, 1995.

[74] Shmuel Sagiv, Thomas W. Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theor. Comput. Sci.*, 167(1&2):131–170, 1996.

[75] A. Salcinau. *Pointer Analysis for Java Programs: Novel Techniques and Applications.* PhD thesis, MIT, 2006.

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

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

[78] J. Whaley and M. Rinard. Compositional pointer and escape analysis for java programs. In *ACM Sigplan Notices*, volume 34, pages 187–206. ACM, 1999.

[79] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for c programs. In *Proceedings of Programming Languages Design and Implementation*, 1995.

[80] Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. *Proceedings of Principles of Programming Languages*, 40(1):351–363, 2005.

[81] K. Yi and S. Ryu. Towards a cost-effective estimation of uncaught exceptions in SML programs. In *Proceedings of the International Symposium on Static Analysis*, pages 98–113, 1997.

[82] G. Yorsh, E. Yahav, and S. Chandra. Generating precise and concise procedure summaries. *Proceedings of Principles of Programming Languages*, 43(1):221–234, 2008.