UNIVERSITY OF CALIFORNIA
SANTA CRUZ

# PROGRAM INCONSISTENCY DETECTION: UNIVERSAL REACHABILITY ANALYSIS AND CONDITIONAL SLICING

A dissertation submitted in partial satisfaction of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

**Aaron Tomb**

June 2011

The Dissertation of Aaron Tomb
is approved:

---

Professor Cormac Flanagan, Chair

---

Professor Martín Abadi

---

Professor Luca de Alfaro

---

Professor Stephen N. Freund

---

Tyrus Miller
Vice Provost
Dean of Graduate Studies

# Contents

# List of Figures

# List of Tables

## Abstract

Program Inconsistency Detection: Universal Reachability Analysis and Conditional Slicing

by

Aaron Tomb

Software inevitably contains mistakes and operates incorrectly in at least some situations. As a result, many techniques and tools have arisen to detect problems in software. However, these techniques have not been widely applied, partly because they tend to work least well in the most common case: on large software systems that have only partial specifications describing correct behavior — often a collection of independent assertions sprinkled throughout the program.

Recent research has suggested that a large class of software bugs fall into the category of *inconsistencies*, or cases where two pieces of program code make incompatible assumptions. Existing approaches to inconsistency detection have used intentionally unsound techniques aimed at bug-finding rather than verification. In this dissertation, we describe an inconsistency detection analysis that extends previous work and is based on the foundation of the weakest precondition calculus. In the ideal case (a completely closed program and a perfect theorem prover), this analysis can serve as a full safety verification technique, while in realistic cases (where some code is unknown and theorem provers are incomplete) it can serve as bug finding technique with a low false-positive rate. It smoothes the path from bug-finding to verification by using the same base analysis for both goals.

Our analysis detects inconsistencies that come as violations of a property we call *universal reachability*. These violations can range from serious crash-causing bugs or security vulnerabilities to dead code, but usually indicate problems that should be fixed. We have applied our analysis to a large body of widely-used open-source software, and found a number of bugs, with few false positives.

To Rebecca, for her unwavering support throughout.

## Acknowledgments

I am grateful for the opportunities that the University made available, especially the chance to work with Cormac Flanagan, whose guidance, perspective, and thorough reviewing as my advisor made all this possible. I am also grateful to Martín Abadi, Luca de Alfaro, and Stephen Freund for agreeing to serve on my committee and for making room in their busy schedules for me. I know their time is valuable and I feel privileged to be in a position to benefit from their experience. I also received valuable support from the National Science Foundation, under award CCR-0341179, from Microsoft's Phoenix Research program, and from UC MICRO, without which this would have been a much greater struggle.

In addition to what the University provided, I am thankful for the opportunity I was given to intern with Willem Visser at NASA Ames Research Center, where I developed valuable research skills that helped my work immeasurably.

I also appreciate the generosity and flexibility shown to me by my fabulous employer, Galois, which provided me the accommodating schedule I needed while finishing the research described herein.

Finally, I owe a great debt to Rebecca Flaum for her emotional support and impeccable editing skills. I doubt I would have been able to complete this dissertation without her consistent encouragement.

# Chapter 1

# Introduction

Even the simplest software is almost guaranteed to contain bugs. Human programmers inevitably have trouble holding in mind every detail of a program's components and their many potential interactions. The time pressures of software development often exacerbate this problem, leading to an initial, rapid push to get *something* written, followed by a clean-up phase in which the developers attempt to improve the original codebase. Tools to verify code and help find bugs can be valuable in this second phase, but many existing tools, especially those that require information about the program's intended behavior, prove to be of limited use.

The problem occurs because specifications added during typical software development often take the form of independent assertions scattered throughout a program, rather than a single, coherent and interrelated contract. When testing is the primary form of quality assurance, these assertions can work well. The compiler can insert a run-time check for each assertion, which will typically cause the program to abort if the assertion is false during an actual execution.

The primary difficulty with testing as an approach to determine whether a program operates correctly is that it has *low coverage*. Only a finite, and usually small, number of paths through the program can be executed feasibly. In practice, many real programs have an effectively infinite state space, so the prospect of ensuring that they behave correctly for all inputs is impractical. To add to this, programmers tend to write tests that are biased toward correct behavior, and leave out tests related to exceptional behavior.

Static program analysis tools can address the problem of coverage through mathematical reasoning about *all possible* program executions. However, when automated verification techniques are applied naïvely to programs that were not designed with verification in mind, they typically yield large numbers of spurious warnings because they lack sufficient information, computational power, or algorithmic sophistication to prove the necessary verification conditions. These warnings tend to be unhelpful because they indicate only failure to prove the absence of defects, rather than successful proof of the presence of defects.

## 1.1 Thesis Overview

This dissertation shows how partial specification information in the form of assertions can be useful for static defect detection without incurring large numbers of false positives and while remaining scalable. While the approach described will not detect all errors that other tools can uncover, it will detect an important class of errors we will refer to as *inconsistencies*, with a low false-positive rate. In addition, once specification information becomes sufficiently comprehensive, the same implementation can be used for full verification.

Broadly, an inconsistency is any case in which an operation at one point in a program depends on one assumption to ensure correct operation, a second operation at a different point depends on another assumption, and these two assumptions are in some way incompatible. This idea has probably existed in the minds of programmers for decades, but was popularized by Dawson Engler in 2001 [39]. The idea was later successfully applied by Dillig *et al.* to find potential null pointer dereferences [37].

Our analysis, which we call *universal reachability analysis* is a novel approach to inconsistency detection based on applying the weakest precondition operator to several modified versions of an input program. For a given program fragment, containing $n$ two-way conditionals, we generate $2n$ *slices*. Each slice is a variant of the initial fragment in which a single conditional is constrained to take only one of its two possible paths. One key characteristic of this approach is that every program point is unconditionally executed in at least one slice, unless it is unreachable.

Once slices have been generated. our analysis checks that each slice can, for at

least one input, terminate without violating any assertion, using a predicate transformer based on the weakest precondition calculus.

Our thesis is that the use of weakest precondition analysis to the problem of inconsistency detection has advantages not present in other approaches. In particular, the ability to use the same implementation for both bug-finding and verification, depending on the completeness of the existing specifications, is unique.

## 1.2 Example

As an example, consider the following C implementation of binary search where the second recursive call has the third and fourth arguments in the wrong order:[1]

```
int bs(int a[], int n, int beg, int end) {
    int mid = (beg + end) / 2;
    if(a[mid] == n) {
        return mid;
    } else if(a[mid] < n) {
        return bs(a, n, mid, end);
    } else if(a[mid] > n) {
        return bs(a, n, mid, beg);
    }
}
```

This program has three slices. One is the sequence of statements that is executed if the first branch of the `if` statement is taken (*i.e.*, when a[mid] == n). The second slice is the sequence that is executed when a[mid] < n. The last is the sequence executed when a[mid] > n.

Now let us extend the language with an `assume` statement, using the standard semantics from Dijkstra's Guarded Command Language [36]. We use `assume` statements to describe what conditions are imposed by each slice. One of the slices of the function body is as follows:

[1] It also, incidentally, has an integer overflow bug in the calculation of mid which our analysis would not detect.

```
int mid = (beg + end) / 2;
assert(beg < end);
assume(a[mid] ≠ n);
assume(a[mid] ≥ n);
assume(a[mid] > n);
return bs(a, n, mid, beg);
```

This slice is guaranteed to crash because the recursive call is always invoked with beg < mid, which will cause the precondition of the callee to be violated. Because the final `else` branch is taken only in the case where the first two branches are not taken, we include assumptions that the conditions of the previous branches are false, along with the assumption that the condition of the current branch is true. These first two branches are redundant in this case, but will not be in cases where the branches are not mutually exclusive.

Another slice for this function is for the (impossible) case where the `else` branch of the final `if` statement is taken:

```
int mid = (beg + end) / 2;
assert(beg < end);
assume(a[mid] ≠ n);
assume(a[mid] ≥ n);
assume(a[mid] ≤ n);
```

This slice is benignly inconsistent; it contains dead code. If the function were written so that the last condition was simply

```
} else {
    return bs(a, n, mid, beg);
}
```

instead of

```
} else if(a[mid] > n) {
    return bs(a, n, mid, beg);
}
```

then this benign inconsistency would not occur. The other slices of this function body are all consistent.

Though this second inconsistency will not cause a crash, we argue that it still should be fixed. If the programmer had included code after the sequence of conditionals, it would never be executed. However, the existence of the code would indicate a mistaken assumption by the programmer that it was reachable. The failure to execute code that was intended to execute is almost certainly a problem.

## 1.3 Alternative Approaches

Many techniques exist that can, in principle, detect many of the same bugs as inconsistency analysis, given sufficient information. Here we describe some of those techniques and explain the cases where inconsistency analysis is a more useful approach.

### 1.3.1 Testing

By far the most widely practiced and popular approach to software quality assurance is testing. It involves directly running an executable program on selected inputs, or in a selected environment, and observing whether its output and effects are as intended.

Testing has several advantages. It can be used on any executable software, with minimal special expertise; it works well with independent assertions; and it only detects assertion failures that can actually occur during execution.

As alluded to above, however, testing has significant disadvantages, as well. One is low coverage. It can detect errors only on code paths that are actually executed, and test suites developed by programmers often focus on ensuring that behavior is correct during normal cases, not that exceptional situations are treated properly.

Another disadvantage is the effort required to develop tests. Though tools can help to some extent, the process of creating tests and describing the expected output can be time-consuming (and therefore expensive). High-coverage test suites can easily be a significant fraction of the size of the programs they test, and require a significant fraction of the development effort.

A consequence is that, while testing is valuable, it is not sufficient. There are, however, some extensions to basic testing that improve its effectiveness, as we describe below.

#### 1.3.1.1 Random Testing

Random testing [29, 20] refers to the case where a separate program automatically generates a random set of test inputs, rather than relying on human input.

Automatic test case generation reduces testing effort. In some cases, it can also improve coverage by making it possible to generate a much larger set of test cases. However, random generation also has the disadvantage of tending to run large numbers of essentially equivalent and potentially uninteresting tests, often failing to test the most important properties. Universal reachability analysis, on the other hand, considers each slice only once and is completely automatic.

#### 1.3.1.2 Concolic Testing

Concolic testing [87] improves on random testing through program instrumentation. It begins by instrumenting the program under test to log every branch it takes, and the values of key variables at that point. Then it runs the program with random inputs.

After a run is complete, a concolic tester examines the log and considers which branches were taken. It then uses constraint solving techniques to determine another set of inputs that will cause the program to make a different choice at one of the branches. This process is repeated until either all branches have been taken, or a time limit is exceeded.

A concolic tester will, if allowed to run to completion, detect some inconsistencies (*i.e.*, violations of universal reachability). However, it can be used only on programs that can be executed, and for which full source code is available, limiting its utility on libraries or partial programs. In addition, concolic testing will not detect inconsistencies that do not cause assertion failures.

### 1.3.2 Ad-hoc Bug Finding Tools

A number of tools exist to search for runtime errors by either looking for particular error-prone programming patterns, or by using simple static techniques such as dataflow analysis. Examples include Coverity Prevent [27], Fortify 360 [91], Klockwork Insight [63], GrammaTech Code Sonar [51], FindBugs[58], and Splint [42].

These tools tend to find many bugs, but also produce many spurious warnings, due to imprecision in their models of program behavior. These extra warnings can often greatly outnumber the reports of true problems, overwhelming their users and greatly reducing their effectiveness on large projects. Our goal is to continue to find real bugs while reducing the number of spurious warnings.

### 1.3.3 Modular Axiomatic Verification

Reasoning based on axiomatic semantics has been widely applied to the problem of software defect detection. Many tools, such as ESC/Java [47], Spec# [12] (specifically, its Boogie tool [10]), Why [43], and many others, use weakest precondition calculation in an attempt to verify (with a greater or lesser degree of soundness) that programs meet their specifications.

As part of an effort to scale up to large programs, these tools operate on one function or method at a time. When they encounter calls in code under analysis, they insert the provided specification for the callee, rather than analyzing the callee itself. The callee will be analyzed, but separately, taking only its provided pre- and postconditions into account.

Modular verification tools can be applied to unannotated programs as defect-detection tools. However, in this mode, they tend to generate huge amounts of output, describing how they have failed to prove a wide range of verification conditions that would be necessary to guarantee that the program under consideration operates correctly. The problem is that most of these warnings do not indicate bugs, but rather that the supplied specifications (which may be minimal or non-existent) do not provide enough information to ensure the absence of bugs.

Our goal in this dissertation is to show that the techniques applied in verification are useful for finding bugs, even in the face of these difficulties, if we interpret

their results appropriately.

In universal reachability analysis, we generate verification conditions (VCs) and failure conditions (FCs) for various modified versions of the original program. If a failure condition is valid, then the associated code is guaranteed to be correct with respect to the known specifications. However, only if an FC is valid does universal reachability analysis report an error. When considering only function FCs, this is a weak defect-detection mechanism. Any function that has been successfully executed, with any inputs, will have a non-valid FC.

However, a valid slice FC can indicate more subtle bugs. Comparison between a pointer variable and the null pointer, for instance, indicates an assumption that the variable can, in some cases be null. The existence of one comparison suggests that the pointer should always be checked against null before being dereferenced, and that any failure to perform this check is likely an error.

### 1.3.4 Model Checking

Model checking is an effective technique for both verification and bug finding that has been practically applied to many industrial systems [22]. In the abstract, model checkers construct (explicitly or implicitly) a graph of all possible states of a program. They then explore this state space and check that each reachable state meets the required specifications, or that certain required states are eventually reached.

Given a closed program, model checkers can effectively check for errors in most programs, even those with minimal specification information. However, to avoid false positives, they need either comprehensive component (*e.g.*, function) specifications, or to analyze the entire program at once.

Unfortunately, complete component specifications rarely exist, and whole-program analysis does not scale well. Therefore, in practice, model checking is most useful either for small or well-specified programs, or for more constrained domains such as hardware verification. Our goal is to handle large, poorly specified software systems.

Techniques like Counterexample-Guided Abstraction Refinement (CEGAR) can significantly improve the scalability of model checking [21], but extremely large software systems still present difficulty. For instance, in 2008, Holzmann, Joshi, and

Groce set forth a "grand challenge" for model checking (and software verification in general): a verifiable file system for non-volatile memory [57]. By software standards, this is likely to be a fairly small system.

As with testing, model checking works most effectively if all source code is available. A model checker can guarantee that all paths from entry to termination of a program are safe, given that those points are known. But for analysis of open programs, such as libraries, model checking is less applicable.

## 1.4 Benefits of Universal Reachability

As the last several sections have suggested, universal reachability analysis has several attractive attributes that position it as a valuable addition to the arsenal of software defect-detection tools.

First is that it has a low false-positive rate. In theory, universal reachability analysis should generate no false positives, and warn only about true inconsistencies that indicate either possible crashes or dead code. Given a perfect decision procedure that can interpret all of the generated verification and failure conditions precisely and given a perfect model of the source language, this ideal would be met. In practice, engineering compromises lead to approximations in some cases. In addition, some true inconsistencies are not serious defects. Inconsistencies can merely indicate dead code, often in the form of explicit checks for situations that can never occur.

In our experiments, the rate of false positives that the analysis reported due to imprecision was fairly low, but the rate of unnecessary checks it reported was fairly high. We consider these unnecessary checks to be bad programming style, and fixing code to avoid them is feasible and likely advantageous when universal reachability analysis is used on a programming project from the start. Our experimental results also show, however, that the number of warnings generated when analyzing large, existing bodies of software, indicating true but relatively uninteresting inconsistencies, can be overwhelmingly large.

Second, in addition to its low false-positive rate, it can be used on partial programs — even programs that cannot be executed. It can be useful for analysis of library code, separate from any specific client. Alternatively, it can be used as part of

an auditing process if source code is provided only as semi-complete evidence that the software does what is intended, rather than as a means for rebuilding the software.

A third benefit of universal reachability is the freedom to adjust computational resource usage. At any point in the analysis, giving up is a sound (though incomplete) option. If generating a failure condition takes too long, solving the failure condition times out, or the number of slices in a function is too large, there are many options. It is possible to simply ignore a particular slice or a whole function, or to approximate the failure condition before attempting to solve it.

In the presence of approximations or timeouts, verification is no longer possible, but inconsistency detection still is. Some inconsistencies may be missed, but any inconsistencies found have all of the guarantees that they would in the ideal case.

The ability to approximate is connected to universal reachability's most significant shortcoming, however. The analysis is not guaranteed to detect all possible assertion violations, only those that are guaranteed to occur as a consequence of passing through any single program point. Though this is a disadvantage, it also makes the analysis particularly appropriate as a complement to testing, perhaps as a technique to guarantee coverage of code that deals with exceptional circumstances that test cases may not cover.

A final point is that, from one perspective, universal reachability can detect only unreachable code. Its capability for detecting bugs comes from its treatment of assertions as conditionals (including implicit assertions that arise as safety preconditions of primitive operations). Assertions give rise to inconsistencies when the success path is unreachable and the failure path is therefore guaranteed.

It may seem that detecting unreachable code is a minor benefit, but we argue that sections of unreachable code are defects. They indicate a misunderstanding on the part of the programmer, and may point to subtle failures in other areas, even if they cannot directly cause misbehavior. In fact, one of the standards for reliable software, DO178B, used by the Federal Aviation Administration in the United States, requires that all code be reachable, and that both paths of any conditional be meaningful and executable [59].

In summary, universal reachability analysis has a number of attractive benefits.

Each is shared with at least one other analysis, but we are not aware of any other technique that can claim them all. Because dynamically checked assertions are a simple and well-adopted way to add specifications to a program in an ad-hoc way, it is important to have analysis tools that can make effective use of the information they provide, in a scalable way, without overwhelming its users with false positives.

## 1.5 Roadmap

The remainder of this dissertation is structured as follows. Ch. 2 provides some background information on program semantics and the weakest precondition calculus, a necessary building block for our approach to inconsistency detection. Ch. 3 then describes the technical details of universal reachability analysis, a technique for detecting program inconsistencies, building on weakest precondition analysis over a variant of Dijkstra's Guarded Command Language. This leads into a description of how universal reachability analysis can be applied to unstructured programs in Ch. 4. We conclude our theoretical discussion with several extensions to universal reachability analysis (Ch. 5) that can improve its bug-finding ability and reduce the number of uninteresting inconsistencies it detects.

Ch. 6 begins the empirical portion of this dissertation, and details our concrete implementation of universal reachability analysis for C programs, including details of how we extend the analysis to go beyond the features of Guarded Commands and support the vagaries of a real-world language. Ch. 7 then discusses the results of applying this implementation to both a collection of widely-used open source programs and a set of benchmarks designed to test the capabilities of bug-finding tools.

To conclude, we discuss the details of related work in Ch. 8, describe some promising future work in Ch. 9, and summarize our contributions in Ch. 10. Finally, Appendix A through Appendix F include proofs for the theorems presented earlier in the dissertation.

# Chapter 2

# Background: Program Semantics

Having described the intuitive notion of program inconsistencies and their uses, we now provide the background necessary to make this intuition formal by providing precise semantics of a programming language. This chapter is provided to aid understanding of the later chapters and is not original research. The next chapter then builds on the language described below to provide a formal description of universal reachability analysis.

Several techniques are in common use to provide precise meanings to programming language constructs. The most widely used are axiomatic semantics, denotational semantics, and operational semantics. Each has strengths and weaknesses, and the best choice of technique depends on the application. We will focus on axiomatic and operational semantics to define the meaning of a simple language on which we can perform universal reachability analysis.

Axiomatic semantics arose from the work of Dijkstra, Floyd, and Hoare in the 1960s and 1970s [36, 49, 56]. They described the meanings of statements in simple imperative languages in terms of the relationship between program state before and after their execution. This approach is particularly fitting for the purposes of reasoning about programs (as opposed to building compilers, for instance) because it provides a way to state and prove facts about programs without over-constraining their meaning with a specific model. Therefore, this chapter provides an overview of axiomatic semantics and the related topic of the weakest precondition calculus as a foundation for universal reachability analysis.

Operational semantics has existed at least since the original development of the $\lambda$-calculus [19] and describes the meanings of programs in terms of a sequence of computational steps on some hypothetical machine. While operational semantics has a less direct connection to logic and reasoning, it is an intuitive description of the way a program may execute on a computer, and can often be easier to validate "by inspection" to determine whether it has the properties one intends.

## 2.1 Language Description

To begin our discussion of axiomatic semantics, we present in Fig. 2.1 the syntax of a programming language we call $GC_{\not\leftrightarrow}$ that we will use throughout the rest of this dissertation as a platform for describing the meaning of universal reachability. The $GC_{\not\leftrightarrow}$ language is a first-order imperative language similar in character to the languages considered by Dijkstra, Floyd, and Hoare, but includes some additions (such as exceptions and procedures) and modifications motivated by more recent research [76, 75, 78].

### 2.1.1 Preliminaries

The concepts of *variables*, *values*, and *stores* are central to the understanding of $GC_{\not\leftrightarrow}$, so we describe their meaning before moving on to the specifics of the syntax. Variables (written $x$, $y$, or $z$) are unique identifiers drawn from an infinite supply. The specific values that programs are able to compute are left unspecified in the language, for generality, and we refer to arbitrary values with the metavariable $v$. In practice, they might consist of entities such as integers, real numbers, or arrays of other values.

A program store, $\theta$, is a functional mapping used to describe the current values of program variables in a particular concrete execution. As a program proceeds, it may change the value of variables described by the store. Because stores are functional mappings, we use the standard functional update operation to model these changes:

$$\theta[x \mapsto v] \stackrel{\text{def}}{=} \lambda x'. \begin{cases} v & x' = x \\ \theta(x) & x' \neq x \end{cases}$$

*Predicates* (written $P$, $Q$, $N$, $X$, or $W$) are also functional mappings, but from stores to Boolean truth values. A predicate can be seen as the characteristic or indicator function that describes the subset of all possible stores that satisfy a particular condition. As we will see below, we can use pairs of predicates to describe how the execution of a program statement affects the contents of the store. We can overload some of the standard operations on predicates to work on characteristic functions as follows:

$$
\begin{aligned}
P \wedge Q &\stackrel{\text{def}}{=} \lambda\theta.\ P(\theta) \wedge Q(\theta) \\
P \vee Q &\stackrel{\text{def}}{=} \lambda\theta.\ P(\theta) \vee Q(\theta) \\
P \Rightarrow Q &\stackrel{\text{def}}{=} \lambda\theta.\ P(\theta) \Rightarrow Q(\theta) \\
\texttt{true} &\stackrel{\text{def}}{=} \lambda\theta.\ \texttt{true} \\
\texttt{false} &\stackrel{\text{def}}{=} \lambda\theta.\ \texttt{false} \\
\mathit{lfp}(\lambda P.\ Q) &\stackrel{\text{def}}{=} \text{the smallest } P' \text{ such that } (\lambda P.\ Q)\ P' = P' \\
P[x := e] &\stackrel{\text{def}}{=} \lambda\theta.\ P(\theta[x \mapsto e(\theta)]) \\
\forall x.\ P &\stackrel{\text{def}}{=} \lambda\theta.\ \forall v.\ P(\theta[x \mapsto v])
\end{aligned}
$$

In the definition of least fixpoint (*lfp*), we assume an ordering on predicates such that $P \sqsubseteq Q$ if and only if $\forall\theta.P(\theta) \Rightarrow Q(\theta)$.

The substitution operation (written $P[x := e]$ and used by the assignment statement we describe below) makes use of *expressions* (written $e$). In real programs, these will typically consist of arithmetic or logical operations, for instance, but we leave them abstract. For the purposes of our discussion, they are simply mappings from stores to values. We will, however, use a variety of concrete expressions in examples, drawn from well-understood theories such as integer arithmetic.

### 2.1.2 Program Structure and Informal Semantics

The syntax for $GC_{\leftrightarrow}$ is shown in Fig. 2.1. It supports familiar language features such as assignment (written $x := e$) and sequencing (written $s_1\ ;\ s_2$). The syntax makes a distinction between atomic instructions and statements that may affect control flow (for symmetry with an unstructured language we introduce later in Ch. 4).

A program is a collection of functions, each of which consists of a name, a precondition annotation $P$, a normal postcondition annotation $N$, an exceptional post-

| | | | | |
|---|---|---|---|---|
| *Program* : | $p$ | ::= | $d^*$ | |
| *Function* : | $d$ | ::= | `fun` $f()\ P\ \{\ s\ \}\ N\ X$ | |
| *Instruction* : | $i$ | ::= | $x := e$ | assignment |
| | | \| | `assert`$(P)$ | dynamic assertion |
| | | \| | `assume`$(P)$ | static assumption |
| | | \| | $f()$ | function call |
| | | \| | `havoc`$(x)$ | arbitrary assignment |
| *Statement* : | $s$ | ::= | $i$ | instruction |
| | | \| | $s\ \square\ s$ | nondeterministic choice |
| | | \| | `skip` | normal termination |
| | | \| | $s\ ;\ s$ | sequence |
| | | \| | `raise` | exceptional termination |
| | | \| | $s\ !\ s$ | exception handler |
| | | \| | $s^*$ | loop |
| *Variable* : | $x, y, z$ | | | |
| *Value* : | $v$ | ::= | $\ldots$ | abstract |
| *Store* : | $\theta$ | : | *Variable* $\rightarrow$ *Value* | |
| *Expression* : | $e$ | : | *Store* $\rightarrow$ *Value* | |
| *Predicate* : | $P, N, X, W$ | : | *Store* $\rightarrow$ *Boolean* | |
| *Context* : | $C$ | ::= | $\bullet$ | hole |
| | | \| | $C\ ;\ s \mid s\ ;\ C$ | sequencing |
| | | \| | $C\ !\ s \mid s\ !\ C$ | exception handling |
| *Evaluation Context* : | | | | |
| | $E$ | ::= | $\bullet \mid E\ ;\ s \mid E\ !\ s$ | |

**Figure 2.1:** Syntax for $GC_{\leftrightarrow}$

condition annotation $X$, and a body $s$. The annotations $P$, $N$, and $X$ represent the claim that when invoked in a state satisfying $P$, the body $s$ will evaluate either to `skip` in a state satisfying $N$ or to `raise` in a state satisfying $X$. We use the global variable $p$ in the following to refer to the function under consideration, to avoid including it as an extra parameter to every definition. We will then say "the program contains `fun` $f()\ P\ \{\ s\ \}\ N\ X$" or $(\texttt{fun}\ f()\ P\ \{\ s\ \}\ N\ X) \in p$ to refer to definitions in the current program.

Intermediate specifications within function bodies are encoded using static assumptions (written $\texttt{assume}(P)$) and dynamic assertions (written $\texttt{assert}(P)$). An assumption trims the execution state space to that in which $P$ is true, providing a way to encode conditional execution. An assertion is a runtime check, and the program will typically abort with an error message if $P$ is not true during a concrete execution. For either an assertion or assumption statement, $P$ will be true after the execution of the statement, if execution proceeds. We also have a generalization to assignment: the $\texttt{havoc}(x)$ instruction assigns an arbitrary, unknown value to $x$. The primary purpose of this instruction is to help support our analysis of loops and procedure calls.

Alternative paths of execution are encoded using the non-deterministic choice operator (written $s_1\ \Box\ s_2$). This operator chooses arbitrarily to execute either $s_1$ or $s_2$. The "if" construct found in most languages can be encoded using a combination of assumption and non-deterministic choice.

$$\texttt{if}\ P\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \stackrel{\text{def}}{=} (\texttt{assume}(P)\ ;\ s_1)\ \Box\ (\texttt{assume}(\neg P)\ ;\ s_2)$$

Iteration is encoded using the loop operator (written $s^*$). This operator executes the statement $s$ an arbitrary number of times. Loops that terminate when a particular condition (or its negation) occur, as found in most programming languages, can be encoded using assumptions.

$$\texttt{while}\ P\ \texttt{do}\ s \stackrel{\text{def}}{=} (\texttt{assume}(P)\ ;\ s)^*\ ;\ \texttt{assume}(\neg P)$$

We include procedure definitions and procedure calls so that we can reason about procedure specifications. For simplicity in the formal semantics, all variables are global; there are no local variables or arguments. Locals and arguments (and most of the rest of the C language) are supported in the implementation described in Ch. 6, however.

Finally, $GC_{\not\rightarrow}$ supports exceptions and recovery from exceptions. The `raise` statement throws an exception. The statement $s_1$ ! $s_2$ executes $s_2$ if $s_1$ evaluates to `raise`. Note that exceptions are not the same as assertion failures. We consider any failed assertion to indicate a bug in the program, whereas a program that reduces to `raise` is acceptable. Sec. 2.4 provides a precise definition of the meaning of different forms of termination.

In some cases, such as the slicing function used for universal reachability analysis, it can be useful to describe statements within the context of a larger, enclosing statement. For this, we define two forms of statement context in Fig. 2.1, normal contexts, $C$. and evaluation contexts, $E$, that contain a hole where a sub-statement can fit. A normal context can be empty ($\bullet$), can refer to either substatement of a sequencing statement ($C$ ; $s$ or $s$ ; $C$), or can refer to either substatement of an exception handler ($C$ ! $s$ or $s$ ! $C$). Evaluation contexts are the subset of normal contexts in which the statement filling the hole is in a position where it can be immediately evaluated. Therefore, evaluation contexts can be empty or refer to the first substatement of a sequencing or exception handling statement.

## 2.2 Axiomatic Semantics

We describe the precise meaning of each of the statements above in terms of how they affect predicates on the store. In general, the approach of axiomatic semantics uses a notation called the *Hoare triple* [56] to describe the possible store changes a statement may cause, where $\{P\}\ s\ \{N\}$ means that if the store satisfies the predicate $P$ when the statement $s$ begins execution, then the store will satisfy $N$ when $s$ has completed (if it does complete). We call $P$ a precondition of $s$, and $N$ a postcondition.

The $GC_{\not\rightarrow}$ language we have presented includes a statement to raise exceptions (`raise`), and one to handle exceptions ($s_1$ ! $s_2$). The inclusion of these constructs requires that our axiomatic semantics also consider the case of exceptional termination. Therefore, we extend our Hoare triples to the form $\{P\}\ s\ \{N, X\}$, where $N$ is the *normal postcondition*, which must hold if $s$ completes execution normally, and $X$ is the *exceptional postcondition*, which must hold if $s$ completes execution by raising an exception. In the following, we will sometimes use the simpler form, $\{P\}\ s\ \{N\}$ when

exceptional behavior is not relevant to the discussion at hand.

For a given statement, many Hoare triples are valid. For instance, the triple $\{\texttt{false}\}\ s\ \{N, X\}$ holds (vacuously) for any statement $s$ and postcondition pair $N, X$, because no initial state can satisfy the precondition `false`. As another example, if $P \Rightarrow P'$ and $\{P'\}\ s\ \{N, X\}$ holds, then $\{P\}\ s\ \{N, X\}$ holds, as well.

## 2.3 Weakest Preconditions and Strongest Postconditions

Given that many Hoare triples are valid for a given statement, which should we use as the canonical description of a statement's meaning? In many cases, an ideal choice is one in which, given a specific predicate for either the pre- or postcondition, we can compute a valid predicate for the other. Said another way, Hoare triples are arbitrary *relations*, in general, but it can be useful to restrict them in such a way that they become *functions* (either from precondition to postcondition or from postcondition to precondition).

If we can compute a postcondition given a precondition, we can reason about how the store changes from a particular initial state. If we can compute a precondition given a postcondition, we can reason about what initial states can ensure that the store satisfies a particular predicate when a statement terminates. The latter choice is often useful when the purpose of the analysis is what precondition is necessary to result in a state that satisfies some intended goal, or to show that the program will always achieve the intended goal, regardless of the initial state.

Two forms of computable Hoare triples are in common use. The *weakest precondition* [36] of a statement $s$, given a postcondition $N$, is a predicate $P$ such that, for all $P'$ for which $\{P'\}\ s\ \{N\}$ is valid, $P' \Rightarrow P$. The *strongest postcondition* of $s$, given a precondition $P$, is a predicate $N$ such that, for all $N'$ for which $\{P\}\ s\ \{N'\}$ is valid, $N' \Rightarrow N$.

Fig. 2.2 gives the axiomatic semantics for $GC_{\leftrightarrow}$ in the form of a weakest precondition transformer. We write the weakest precondition of $s$, given a normal postcondition $N$ and exceptional postcondition $X$, as $wp(s, N, X)$.

The assignment statement is one of the most central components of any imperative programming language, and therefore of central importance to weakest pre-

$wp : Statement \times Predicate \times Predicate \rightarrow Predicate$

| $s$ | $wp(s, N, X)$ |
|---|---|
| $x := e$ | $N[x := e]$ |
| `havoc`$(x)$ | $\forall x'.N[x := x']$ |
| `skip` | $N$ |
| `assert`$(P)$ | $P \wedge N$ |
| `assume`$(P)$ | $P \Rightarrow N$ |
| $s_1$ ; $s_2$ | $wp(s_1, wp(s_2, N, X), X)$ |
| $s_1 \ \square \ s_2$ | $wp(s_1, N, X) \wedge wp(s_2, N, X)$ |
| `raise` | $X$ |
| $s_1$ ! $s_2$ | $wp(s_1, N, wp(s_2, N, X))$ |
| $s^*$ | $lfp(\lambda P.\ \ N \wedge wp(s, P, X))$ |
| $f()$ | $wp(s', N, X)$ |
| | if `fun` $f()\ P\ \{\ s\ \}\ N'\ X' \in p$ |
| | and $s' =$ `assert`$(P)$ ; `havoctargets`$(s)$; |
| | `assume`$(N')\ \square\ $(`assume`$(X')$ ; `raise`) |

**Figure 2.2:** Weakest Preconditions of Statements in $GC_{\leftrightarrow}$

condition analysis. The rule for assignment states that the weakest precondition that can guarantee a postcondition of $N$ for the assignment statement $x := e$ is $N$ with all occurrences of $x$ replaced by $e$ (written $N[x := e]$). A variation on assignment is the `havoc` statement, which assigns an unknown value to its argument. We represent this unknown value with a fresh variable, $x'$, and say that the weakest precondition of `havoc`$(x)$ is $\forall x'.\ N[x := x']$.

The `skip` statement has no effect, and exists both as a placeholder and as the representative of normal program termination. The weakest precondition of the `skip` statement is the normal postcondition $N$, unmodified.

Assertion (`assert`$(P)$) and assumption (`assume`$(P)$) statements are used to explicitly introduce restrictions on program state. An assertion of predicate $P$ conjoins $P$ with the postcondition $N$, which adds the requirement that the postcondition $N$ can only be achieved starting from a state in which $P$ is true. Any state in which $P$ is not true before the execution of (`assert`$(P)$) results in an assertion failure. An assumption `assume`$(P)$ results in a similar but critically different weakest precondition: $P \Rightarrow N$. Therefore, if $P$ holds before execution of an assumption, $N$ must hold, as well.

However, if $P$ does not hold, the assumption places no additional restrictions on the program state. Assumptions can therefore be used to describe conditional execution.

Assumptions are most useful in conjunction with non-deterministic choice. The weakest precondition of $s_1 \ \square \ s_2$ is the weakest precondition of $s_1$ conjoined with that of $s_2$. That is, the initial state must be such that both $s_1$ and $s_2$ can execute. Assumptions within either of these statements can be used to make the choice between the statements conditional on particular predicates. As mentioned before, the conditional statement found in most imperative languages can be modeled as follows:

$$\texttt{if } P \texttt{ then } s_1 \texttt{ else } s_2 \stackrel{\text{def}}{=} (\texttt{assume}(P) \ ; \ s_1) \ \square \ (\texttt{assume}(\neg P) \ ; \ s_2)$$

Statements can be placed in sequence using the notation $s_1 \ ; \ s_2$. To calculate the weakest precondition of a sequenced pair of statements, we first calculate the weakest precondition of the second statement, given postconditions $N$ and $X$. Then we use this weakest precondition as the normal postcondition when calculating the weakest precondition of the first statement (along with the original exceptional postcondition).

Up until this point, the weakest precondition of each of the statements in $GC_{\leftarrow}$ has depended only on the normal postcondition. The `raise` statement is similar to the `skip` statement, except that it passes through the exceptional postcondition $X$ instead of the normal postcondition $N$.

To handle exceptions thrown with `raise`, the catch statement $(s_1 \ ! \ s_2)$ works symmetrically to the sequence statement. It uses the precondition of the second statement as the exceptional postcondition when calculating the precondition of the first.

Loops introduce an additional complexity to the analysis. A loop may iterate an arbitrary number of times, and therefore we need to find some predicate that is true of both the initial and final states of an arbitrary iteration. Any predicate that is a fixpoint of the weakest precondition transformer over the loop body will satisfy the implied Hoare logic for the loop. To calculate the *weakest* precondition, we calculate the *least* fixpoint.

Finally, we calculate the weakest precondition of procedure calls using the specification of the target procedure. To reason about the effects of composite statements such as function bodies, which may include an arbitrary number of assignment or havoc statements, we define a function $targets(s)$ in Fig 2.3 that determines the set

of variables that may be updated when $s$ is evaluated. In the presence of recursion, we will need to calculate the least fixpoint of the $targets(s)$ function, but that will present no trouble, since the targets of any assignment are completely static in $GC_{\leftrightarrow}$.

The main use of the $targets(s)$ function is in conjunction with $\texttt{havoc}(x)$, to obscure the values of the variables assigned by $s$. Therefore, we also introduce an abbreviation.

$$\texttt{havoctargets}(s) \stackrel{\text{def}}{=} \texttt{havoc}(targets(s))$$

$$
\begin{array}{rcl}
targets & : & Statement \rightarrow 2^{Variable} \\
targets(x := e) & \stackrel{\text{def}}{=} & \{x\} \\
targets(\texttt{havoc}(x)) & \stackrel{\text{def}}{=} & \{x\} \\
targets(\texttt{assert}(P)) & \stackrel{\text{def}}{=} & \emptyset \\
targets(\texttt{assume}(P)) & \stackrel{\text{def}}{=} & \emptyset \\
targets(f()) & \stackrel{\text{def}}{=} & targets(s) \\
 & & \text{where } f \text{ is defined as } \texttt{fun } f() \ P \ \{ \ s \ \} \ N \ X \\
targets(\texttt{skip}) & \stackrel{\text{def}}{=} & \emptyset \\
targets(\texttt{raise}) & \stackrel{\text{def}}{=} & \emptyset \\
targets(s^*) & \stackrel{\text{def}}{=} & targets(s) \\
targets(s_1 \ \square \ s_2) & \stackrel{\text{def}}{=} & targets(s_1) \cup targets(s_2) \\
targets(s_1 \ ; \ s_2) & \stackrel{\text{def}}{=} & targets(s_1) \cup targets(s_2) \\
targets(s_1 \ ! \ s_2) & \stackrel{\text{def}}{=} & targets(s_1) \cup targets(s_2)
\end{array}
$$

**Figure 2.3:** Target Variables Written by Statements

Given this description of a statement's effects, a function call will satisfy the postcondition pair $N, X$ if, assuming the execution starts in a state satisfying the function's precondition, the function's normal postcondition implies the desired normal postcondition and the function's exceptional postcondition implies the desired exceptional postcondition, for all possible store modifications that the function body may cause.

Note, however, that this precondition is only a *weakest* precondition if the pre- and postcondition annotations on the function are precise. If they are, for instance, all `true`, then $wp(f(), N, X)$ will over-approximate the possible effects of $f$, and require a stronger precondition to guarantee the given postcondition pair. While our intent is

to support programs that may not have precise annotations, the over-approximation of function effects may lead the analysis of Ch. 3 to miss defects, but will not lead to spurious warnings.

## 2.4 Operational Semantics

In addition to the axiomatic semantics given in the previous chapter, we provide an operational semantics for $GC_{\not\Rightarrow}$, to more naturally reason about the different forms of program termination, and to validate the correctness of later reasoning based on weakest preconditions. Operational semantics for the $\lambda$-calculus were derived as early as the 1930s [19] and have been widely used to describe more complex programming languages since the Algol 68 report [8].

Fig. 2.4 and Fig. 2.5 show a collection of evaluation rules that allow derivation of the state resulting from execution of an instruction or statement, respectively, given an initial state, where a *state* is a combination of a store and a remaining statement to execute.

The key difference between the axiomatic and operational semantics is that the operational semantics describes changes to the store directly and specifically, rather than changes to what predicates the store satisfies. An operational semantics is typically more appropriate as the basis of a language implementation, whereas an axiomatic semantics is more likely to admit multiple implementations, and leave many implementation details unspecified.

Recall that a store $\theta$ is a mapping from variables $x$ to values. We abstract over the exact syntax of expressions and predicates, but define both as functions on stores. Therefore, we can write $e(\theta)$ to denote the evaluation of expression $e$ to a value in store $\theta$, and $P(\theta)$ to denote the truth value of predicate $P$ in store $\theta$. A *state* is a pair $\theta, s$ of a store and a statement. Fig. 2.4 and Fig. 2.5 define the small-step evaluation relation $\theta, s \rightarrow \theta', s'$ on states.

Assertion and assumption statements have no effect on the store. As described in evaluation rules [E-ASSERT] and [E-ASSUME], they each evaluate their associated predicate with respect to the current store and proceed only if the result is `true`.

All modification of the store is ultimately due to either an assignment or havoc

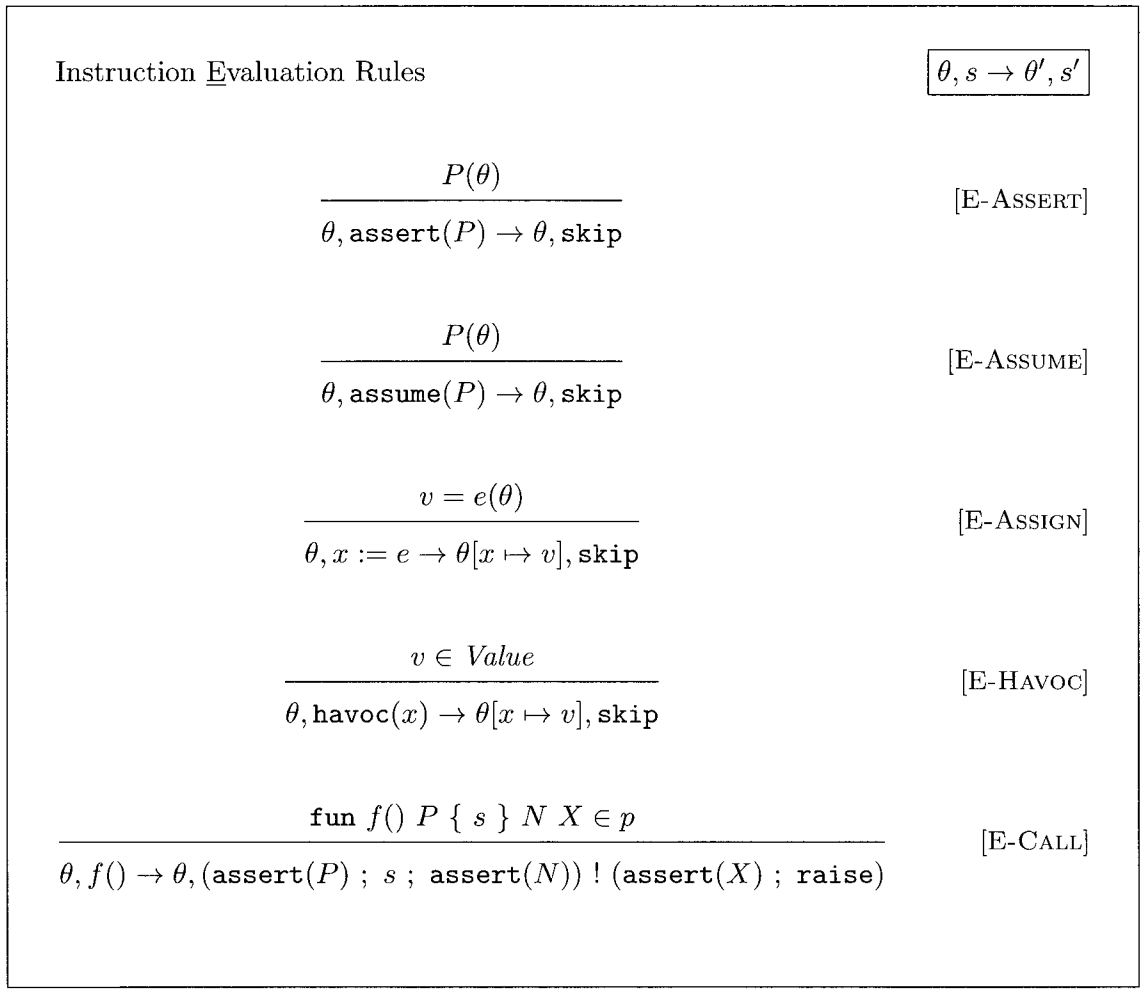


**Figure 2.4:** Operational Semantics of Instructions

statement. The evaluation rule for assignments, [E-ASSIGN], first evaluates $e$ with respect to the current store to yield a value $v$. It then produces a new state with the store updated to map $x$ to $v$. The evaluation rule for havoc, [E-HAVOC], chooses an arbitrary value $v$ and produces a new state with the store updated to map $x$ to $v$.

The evaluation of the choice statement is shown in rules [E-CHOICE1] and [E-CHOICE2]. The former simply executes $s_1$, while the latter executes $s_2$. Either rule may fire, arbitrarily, so the choice statement is non-deterministic.

Evaluation of the sequencing statement proceeds by first evaluating its first sub-statement according to [E-SEQ3]. If the first sub-statement evaluates to `skip`, the second sub-statement is evaluated, as shown in [E-SEQ1]. If, on the other hand, the

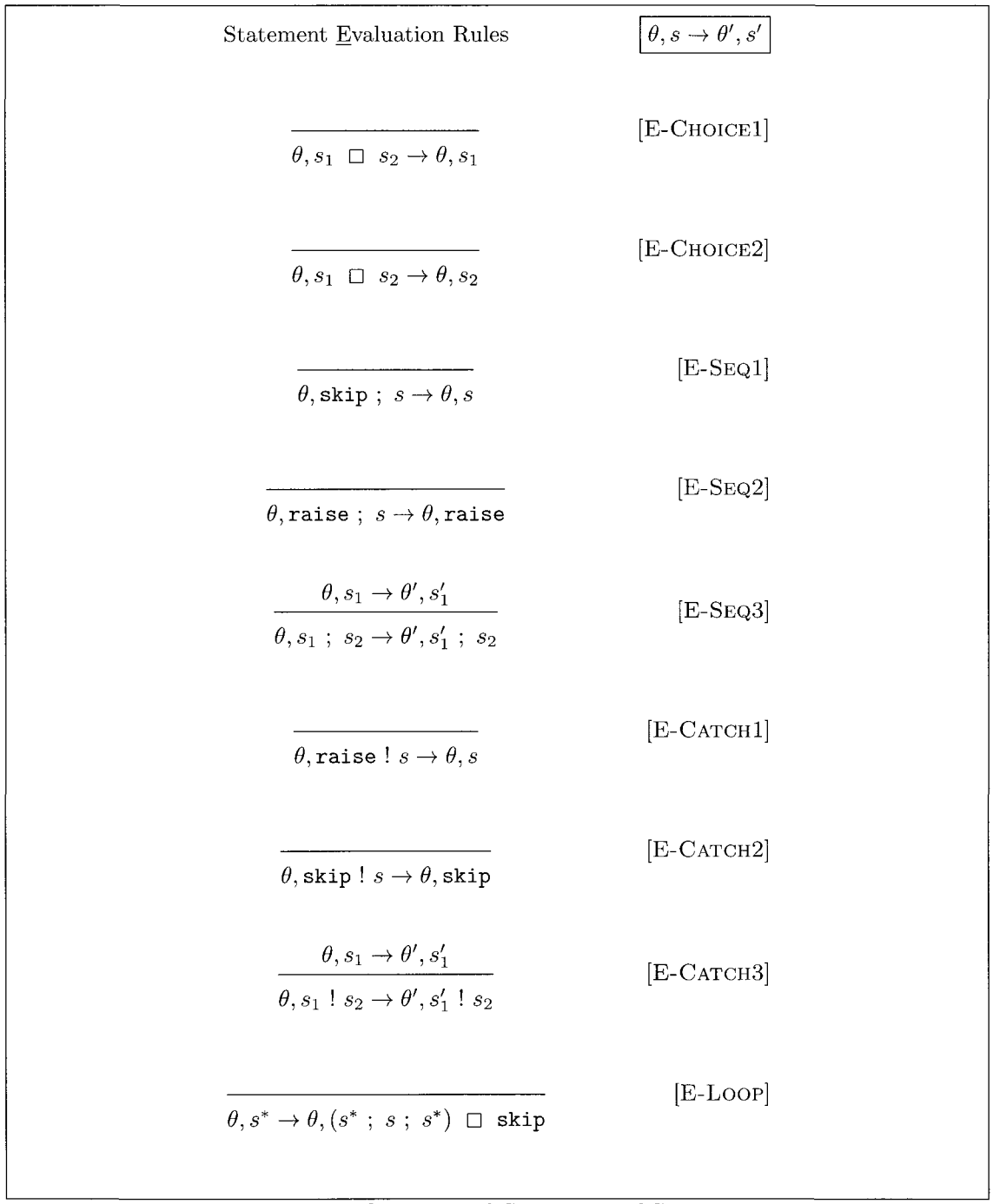


**Figure 2.5:** Operational Semantics of Statements

first sub-statement evaluates to `raise`, the second sub-statement is ignored and the exception is propagated, as shown in [E-SEQ2].

Exceptions are propagated until caught by an exception handler. If the first sub-statement of a catch statement evaluates to `raise`, the second sub-statement is evaluated, as shown in [E-CATCH1]. If the first sub-statement evaluates to `skip`, the second sub-statement is ignored, and the `skip` is propagated, as shown in [E-CATCH2]. Thus, the sequencing and exception handling statements are symmetrical, with evaluation of the first sub-statement handled by [E-CATCH3].

The loop statement is evaluated by repeatedly evaluating the associated statement. The evaluation rule [E-LOOP] transforms a loop statement, $s^*$ into the sequence $s^*$ ; $(s$ ; $s^*)$ $\square$ `skip`, which means that bare loops can execute an arbitrary number of times. Terminating loops can be encoded using assumptions to trim the state space, as shown in the previous section.

Function calls are evaluated by inlining the target function body along with its specification and an exception handler, as shown in [E-CALL]. Specifically, this involves asserting the precondition and evaluating the body. If the body terminates normally, the normal postcondition is asserted. If the body terminates with an exception, the exceptional postcondition is asserted.

Programs can terminate (or fail to terminate) in several ways. If application of the evaluation rules yields a state where the statement is `skip`, the program has *terminated normally.* If the statement is `raise`, the program has *terminated exceptionally.* None of the evaluation rules will allow execution to proceed in either of these cases.

A program is *stuck* if it is in any state where the statement is neither `raise` nor `skip`, and none of the evaluation rules apply. A stuck program can arise in any state where the current statement is an assertion or assumption, and the associated predicate is not true when applied to the current store. If the program is stuck when the current statement is an assertion, we call it an *assertion failure.* Finally, a program *diverges* if there is no finite sequence of evaluation rule applications that will yield an assertion failure, normal termination, or exceptional termination. Divergence includes stuck programs in which the current statement is an assumption.

## 2.5 Arrays

Realistic software almost universally has some structure that is best represented using arrays. At the simplest end of the spectrum, software may use array variables directly. Fixed-size arrays could be modeled using a large number of scalar variables, but this can quickly become awkward and computationally expensive. For variable-size arrays, this approach is not possible. In addition, any program that uses pointers can be modeled by treating the program memory space as an array, and treating pointers as indices into this array. To clarify certain aliasing axioms, and improve the efficiency of automated theorem proving, we may want to treat memory as a collection of separate arrays, instead.

Our initial presentation of the syntax and semantics of $GC_{\rightarrow}$ did not include array operations. However, they are relatively straightforward to model. First, we need to add syntax for an array assignment statement, as shown in Fig. 2.6. With this extension, an assignment may either overwrite an entire array variable $x$, written $x := e$, or simply an element $i$ of the array that $x$ represents, written $x[i] := e$. We also allow havoc statements to operate on either array variables or array indices. That is, for an array variable $x$, we can write either $\texttt{havoc}(x)$ or $\texttt{havoc}(x[i])$. The former replaces the entire array with an unknown value, while the latter only eliminates knowledge of the value of a specific array element.

$$
\begin{array}{rrcl}
\textit{Array}: & a & : & \mathbb{N} \rightarrow \textit{Value} \\
\textit{Statement}: & s & ::= & \ldots \\
 & & | & x[e] := e \\
 & & | & \texttt{havoc}(x[e])
\end{array}
$$

**Figure 2.6:** Array Syntax

To support extending the operational semantics with array operations, we extend the set of values to include array values, which are functions from natural numbers to finite values. We use the same functional mapping semantics for arrays that we do for predicates, expressions, and stores. The array lookup operation is simply function application, and the array update operation is functional update. We leave out specific

syntax for array creation, as it does not play a critical role in our analysis. However, in a concrete instantiation of the language, it would probably occur in some form such as $x := \mathsf{Array}(10)$ or $x := \{1, 20, 5\}$, creating arrays of size 10 and 3, respectively, where the former has undefined contents, and the latter has known contents.

With the addition of array values, we can define the evaluation rule for the array assignment statement that appears in Fig. 2.7.

$$\frac{v_1 = e_1(\theta) \qquad v_2 = e_2(\theta) \qquad \theta(x) \in \mathbb{N} \to \mathit{Value} \qquad v_1 \in \mathbb{N}}{\theta, x[e_1] := e_2 \to \theta[x \mapsto \theta(x)[v_1 \mapsto v_2]], \mathtt{skip}} \quad \text{[E-ArrAssign]}$$

**Figure 2.7:** Operational Semantics for Array Assignment

Finally, we extend the axiomatic semantics (the *wp* transformer) as shown in Fig. 2.8. As in the operational semantics, we treat each array as a single array-valued variable. Though we have treated expression syntax as abstract so far, here we require that there exists an array update expression, which we write $\mathit{update}(x, e_1, e_2)$.

| $s$ | $wp(s, N, X)$ |
|---|---|
| $x[e_1] := e_2$ | $N[x := \mathit{update}(x, e_1, e_2)]$ |

**Figure 2.8:** Weakest Preconditions for Array Assignment

To reason about loops and procedures containing array updates, we extend the $\mathit{targets}(s)$ function to cover array assignment and havoc statements.

$$\begin{aligned} \mathit{targets}(x[e_1] := e_2) &\stackrel{\text{def}}{=} \{x\} \\ \mathit{targets}(\mathtt{havoc}(x[e_1])) &\stackrel{\text{def}}{=} \{x\} \end{aligned}$$

It would be possible to extend the result type of $\mathit{targets}(s)$ to allow more precise reasoning about the effects of loop and procedure bodies. Our experience in practice, however, suggests that the added complexity outweighs the improved precision.

# Chapter 3

# Universal Reachability Analysis

We now formally present an analysis for detecting inconsistencies using an approach we call universal reachability analysis. We then discuss how to use the same analysis for inconsistency detection and verification, bring up the issues that arise in inter-procedural analysis, present several approaches for handling loops, and show a collection of example inconsistency derivations.

## 3.1 Definitions

Now we describe the key property that we use to determine when a statement is inconsistent. We begin our analysis by considering the conditions in which it is possible for a sub-statement $s'$ to begin to be evaluated when evaluating an enclosing statement $s$. For this to happen, it must be possible for $s'$ to find its way into an evaluation context.

**Definition 1** (Weakly Reachable). *Given a statement $s$, the sub-statement $s'$ is defined to be* weakly reachable *in $s$ if there exist stores $\theta$, $\theta'$ and an evaluation context $E$ such that $\theta, s \rightarrow^* \theta', E[s']$.*

Next we consider the case where $s'$ is weakly reachable in $s$, and it is also possible for the entire statement $s$ to be reduced to `skip` or `raise`.

**Definition 2** (Strongly Reachable). *Given a statement $s$, the sub-statement $s'$ is defined to be* strongly reachable *in $s$ if there exist stores $\theta$, $\theta'$, $\theta''$, an evaluation context $E$, and*

*a statement* $s'' \in \{\texttt{skip}, \texttt{raise}\}$ *such that*

$$\theta, s \rightarrow^* \theta', E[s'] \rightarrow^* \theta'', s''$$

Finally, we extend this notion to consider the case where all sub-statements of a given statement are strongly reachable.

**Definition 3** (Universally Reachable). *A statement s is defined to be* universally reachable *if each sub-statement s′ in s is strongly reachable.*

Our ultimate goal is now to show that all of the function bodies in a program are universally reachable. We will say that any statement that is not universally reachable is *inconsistent*. The next section describes the first step in *computing* whether a statement is universally reachable.

## 3.2 Slicing

The previous section defined the property we would like all statements in a program to satisfy, but we have not said anything about how to compute when this property holds, or, more importantly, how to determine when it does not hold. This section describes the first step in that process.

Our analysis computes universal reachability by rewriting an input statement into several versions, or *slices*. Informally, a slice is a modified form of a statement in which one branch of a chosen choice statement is forced to be taken[1]. This modification requires also that any enclosing choice statements are forced to take the path that leads to the chosen choice statement. For every choice statement in $GC_{\leftrightarrow}$ there are two slices. Note that slices are not the same as paths: the number of slices in a program grows linearly with the number of choice statements in a program; the number of paths grows exponentially.

We can formalize the preceding intuitive notion of slicing with the function shown in Fig. 3.1. This function computes all slices, $s_1 \ldots s_n$ of $s$ and inserts each of them into the enclosing context $C$, yielding the collection of slices $C[s_1] \ldots C[s_n]$. Thus, $\mathit{slices}(\bullet, s)$ computes all slices of $s$.

[1]Note that this is a different meaning of the term *slice* than that introduced by Weiser [93].

$$
\begin{aligned}
\mathit{slices} &: \mathit{Context} \times \mathit{Statement} \rightarrow 2^{\mathit{Statement}} \\
\mathit{slices}(C, \imath) &= \{C[\imath]\} \\
\mathit{slices}(C, \texttt{skip}) &= \{C[\texttt{skip}]\} \\
\mathit{slices}(C, \texttt{raise}) &= \{C[\texttt{raise}]\} \\
\mathit{slices}(C, s_1 \ ; \ s_2) &= \mathit{slices}(C[\bullet \ ; \ s_2], s_1) \cup \mathit{slices}(C[(s_1 \ ! \ \bot) \ ; \ \bullet], s_2) \\
\mathit{slices}(C, s_1 \ ! \ s_2) &= \mathit{slices}(C[\bullet \ ! \ s_2], s_1) \cup \mathit{slices}(C[(s_1 \ ; \ \bot) \ ! \ \bullet], s_2) \\
\mathit{slices}(C, s_1 \ \Box \ s_2) &= \mathit{slices}(C, s_1) \cup \mathit{slices}(C, s_2) \\
\mathit{slices}(C, s^*) &= \mathit{slices}(C[(s^* \ ! \ \bot) \ ; \ \bullet \ ; \ s^*], s)
\end{aligned}
$$

**Figure 3.1:** Slice Computation

To slice an instruction, `skip`, or `raise`, in a context $C$, we simply insert the statement into $C$.

To slice a sequence, $s_1 \ ; \ s_2$, we slice $s_1$ in extended context where it is followed by $s_2$. To slice $s_2$, however, we must account for the possibility that $s_1$ evaluates to `raise`, in which case $s_2$ will be skipped. To ensure that $s_2$ will be executed, we slice it in the extended context where it is preceded by $s_1 \ ! \ \bot$, where $\bot$ represents a divergent computation such as `assume(false)`. In the absence of exceptions, we would slice $s_2$ in a context symmetrical with that used when slicing $s_1$

To slice an exception handler, $s_1 \ ! \ s_2$, we use a symmetrical approach. In this case, however, we need to account for the fact that $s_1$ may evaluate to `skip`, in which case the exception handler will be skipped.

To slice a choice statement, $s_1 \ \Box \ s_2$, we slice each of its branches in the same context as the entire statement. This has the effect of discarding one branch in one subset of slices, and discarding the other branch in the other subset of slices.

To slice in the presence of loops, we want to ensure that the selection of a particular choice branch occurs on an *arbitrary* loop iteration. Otherwise, the selection of a particular choice may introduce artificial inconsistencies. Often, loop bodies contain conditional branches that may be possible only on certain loop iterations, such as the first iteration, the last iteration, or alternating iterations.

To understand how we perform slicing at an arbitrary iteration, consider a loop $s^*$ with body $s$. To ensure that a slice of $s$ occurs in an arbitrary iteration of $s^*$,

we first slice $s$ to yield a set of slices. For each of these slices, $s'$, we create an enclosing slice with an arbitrary number of loop iterations occurring both before and after $s'$:

$$s^* \ ; \ s' \ ; \ s^*$$

Before we begin reasoning about slices, specifically, we introduce an intermediate concept, related to universal reachability. We say that a statement is *terminable* if it can be fully reduced to `skip` or `raise`.

**Definition 4** (Terminable). *A statement $s$ is defined to be* terminable *if there exist stores $\theta$, $\theta'$, and a statement $s' \in \{\texttt{skip}, \texttt{raise}\}$ such that*

$$\theta, s \rightarrow^* \theta', s'$$

Though similar in style to universal reachability, this property is easier to check. In particular, it does not refer to sub-statements, but only to the overall statement in question.

Our key theorem is then that we can use terminability of the slices of a statement to decide whether the original statement is universally reachable.

**Theorem 1.** *Given a statement $s$, the following are equivalent:*

1. *$s$ is universally reachable*
2. *$\forall s' \in \mathit{slices}(\bullet, s)$, $s'$ is terminable*

*Proof.* By induction on the structure of $s$. The full proof appears in Appendix A. □

## 3.3 Termination Analysis

It now remains to compute whether or not a statement can possibly terminate, for at least one initial store. Given a set of slices computed for an original statement, we decide the terminability of each slice with a combination of a weakest precondition calculation and an algorithmic decision procedure (typically for a subset of first-order logic, though our abstraction over predicates could allow for a more expressive logic).

The weakest precondition operation we use for termination analysis (Fig. 3.2) differs slightly from the traditional definition shown in Sec. 2.3. We add a third predicate argument representing the postcondition for the case where the statement fails an assertion (or *goes wrong*). as defined in Sec. 2.4, and modify the interpretation of assertions to take this extra postcondition into account.

$wp : \mathit{Statement} \times \mathit{Predicate} \times \mathit{Predicate} \times \mathit{Predicate} \rightarrow \mathit{Predicate}$

| $s$ | $wp(s, N, X, W)$ |
|---|---|
| $x := e$ | $N[x := e]$ |
| `havoc`$(x)$ | $\forall x'.N[x := x']$ |
| `skip` | $N$ |
| `assert`$(P)$ | $(P \wedge N) \vee (\neg P \wedge W)$ |
| `assume`$(P)$ | $P \Rightarrow N$ |
| $s_1$ ; $s_2$ | $wp(s_1, wp(s_2, N, X, W), X, W)$ |
| $s_1 \ \square \ s_2$ | $wp(s_1, N, X, W) \wedge wp(s_2, N, X, W)$ |
| `raise` | $X$ |
| $s_1$ ! $s_2$ | $wp(s_1, N, wp(s_2, N, X, W), W)$ |
| $s^*$ | $lfp(\lambda P.\ \ N \wedge wp(s, P, X, W))$ |
| $f()$ | $wp(s', N, X, W)$ |
| | if `fun` $f()\ P\ \{\ s\ \}\ N'\ X' \in p$ |
| | and $s' =$ `assert`$(P)$ ; `havoctargets`$(s)$; |
| | `assume`$(N')\ \square$ (`assume`$(X')$ ; `raise`) |

**Figure 3.2:** Weakest Preconditions for Universal Reachability on $GC_{\leftrightarrow}$

If the value of $W$, the postcondition for going wrong, is `false` then the treatment of assertions matches that in the traditional definition of weakest preconditions. However, if $W$ is `true`, $wp($`assert`$(P), N, X, W)$ simplifies to $P \Rightarrow N$, and therefore treats assertions with the traditional semantics for assumptions.

If we treat assertions with the semantics of assumptions, we cannot prove that a program will fail an assertion, but we can prove that a program will make either an assertion or an assumption that can never be true in the context where it occurs. An instance of this situation indicates either an assertion that will always fail (a program crash) or a branch that cannot be taken (dead code).

While either of these cases is fairly uninteresting when considering an entire function body, they are more likely to indicate interesting bugs when considering slices.

While a function that always fails during execution would almost certainly be discovered during testing, it may be more difficult to discover a particular branch that will always lead to failure, especially when the conditions that lead to the branch being taken are complex and hard to induce on demand. A significant fraction of the code in typical software systems is devoted to handling errors, and this code is often the least well-tested portion of a system, depending heavily on details of the execution environment [28].

We begin to move toward a formal description of universal reachability computation by connecting our modified axiomatic semantics (the *wp* transformer) to the operational semantics from Sec. 2.4.

We state the theorem in the absence of function calls, because the modular reasoning involved has been heavily studied and is orthogonal to the topic of universal reachability

**Theorem 2.** *If* $wp(s, N, X, W) = P$ *and* $s$ *contains no function calls and there exist stores* $\theta$ *and* $\theta'$ *and a statement* $s'$ *such that* $\theta, s \rightarrow^* \theta', s'$, *where* $s' \in \{\texttt{skip}, \texttt{raise}\}$, *and* $\theta$ *satisfies* $P$ *then one of the following cases holds:*

1. $s' = \texttt{skip}$ *and* $\theta'$ *satisfies* $N$
2. $s' = \texttt{raise}$ *and* $\theta'$ *satisfies* $X$

*Proof.* By induction on the structure of $s$. The full proof appears in Appendix B. □

Next, to decide the terminability of a statement $s$, we can compute the predicate $wp(s, \texttt{false}, \texttt{false}, \texttt{true})$. We call the result of this particular weakest precondition calculation a *failure condition.* A failure condition which is always true indicates that the end of the statement can never be reached.

**Theorem 3.** *For any call-free statement* $s$, *if* $wp(s, \texttt{false}, \texttt{false}, \texttt{true})$ *is valid then* $s$ *is not terminable.*

*Proof.* A direct consequence of Theorem 2. The full proof appears in Appendix C. □

We can now combine the previous three theorems to arrive at the following primary result that states that an algorithmic approach to detecting violations of universal reachability is sound.

**Theorem 4.** *Given a call-free statement $s$, if there exists an $s'$ such that $s' \in slices(\bullet, s)$ and $wp(s', \texttt{false}, \texttt{false}, \texttt{true})$ is valid then $s$ is not universally reachable*

*Proof.* A straightforward consequence of Theorem 1 and Theorem 3. The full proof appears in Appendix D. □

To determine validity automatically, we depend on a decision procedure that is sound (though not necessarily complete). We describe soundness in terms of two auxiliary judgments. First, we use a notion of *validity*.

**Definition 5** (Validity). *A predicate $P$ is* valid, *written $\models P$, if it evaluates to* `true` *in all possible stores.*

Next, we introduce another notation to describe the results of our logical decision procedure.

**Definition 6** (Algorithmic Provability). *A predicate $P$ is* algorithmically provable, *written $\vdash_{alg} P$ if a decision procedure reports that it is valid.*

Finally, we say that a decision procedure is sound if all predicates that are reported valid by the prover are in fact valid.

**Assumption 1** (Prover Soundness).

$$\forall P. \left(\vdash_{alg} P\right) \Rightarrow (\models P)$$

We do not demand that the opposite implication holds: the decision procedure may be incomplete and therefore return a result of "invalid" or "unknown" for a valid formula. If the decision procedure is sound but incomplete, we may miss errors but we will get no false alarms.

Alternatively, it may be possible to make use of a termination analysis of the sort used in Terminator [25, 26]. We leave an investigation of alternative termination checks for future work (although we note that our analysis requires only that we know if a statement *can* terminate for *some* input, not whether it *always* terminates).

## 3.4 Verification

Although we have presented our analysis as a defect-detection mechanism so far, one advantage of using an approach based on weakest preconditions is that it generalizes to full verification when sufficient information is available.

If the specifications present in a program are sufficiently complete that it is possible to prove that $P \Rightarrow wp(s, N, X, \texttt{false})$ is valid for a function $\texttt{fun}\ f()\ P\ \{\ s\ \}\ N\ X$, then we can be guaranteed that it meets its specification. Only if we cannot verify a function does it make sense to apply universal reachability analysis in an attempt to determine whether the function is guaranteed to violate the known components of its specification.

## 3.5 Procedures

In the presentation of universal reachability analysis to this point, procedure calls are sliced intra-procedurally, and the weakest precondition function uses only the provided specification of the target function when analyzing a function call. In the absence of complete procedure specifications, this approach could seem overly imprecise.

To address the imprecision that arises in the lack of procedure specifications, it could be tempting to inline target functions and perform slicing across procedure boundaries. Unfortunately, this can result in large numbers of false positives.

The problem arises because most procedures represent abstractions, intended to be used in a variety of conditions. In the context of any specific function call, only some subset of these conditions is likely to hold, and therefore some portions of the target function body will be dead code. If the function body were inlined, each branch in the target function that could not be taken in one specific context where it is called would be flagged as inconsistent.

The problem of inlining procedures arises when using macros in C, as well. Many macros are treated as small, inline procedures, and they often contain conditionals that will be forced to take a single path in the specific context where they occur. The results in Ch. 7 show that macros are one of the most significant causes of uninteresting inconsistencies.

## 3.6 Loops and Approximation

In Sec. 2.3 we provide a semantics for loops that depends on a fixpoint construct in the logic:

$$wp(s^*, N, X, W) \quad = \quad lfp(\lambda P.\ N \wedge wp(s, P, X, W))$$

This is a precise encoding; the theorems from the previous sections all hold. However, existing automated decision procedures do not typically support fixpoint operations. Since our aim is to support automatic defect detection, we would like a more tractable approximation.

We begin by defining an approximation relation over statements that allows us to say when one the behavior of one statement over-approximates the behavior of another.

**Definition 7** (Statement Approximation). *A statement $s$ over-approximates a statement $s'$, written $s \sqsupseteq s'$, if and only if for all predicates $N$, $X$, and $W$, $wp(s, N, X, W) \Rightarrow wp(s', N, X, W)$.*

This is useful because we use validity checks to detect inconsistencies. If $wp(s, N, X, W)$ is valid, and $s \sqsupseteq s'$, we know that $wp(s', N, X, W)$ is also valid, and that we therefore will not generate spurious inconsistency warnings.

Given a notion of statement approximation, we can then rewrite $s^*$ with an approximate version that guarantees that we get no false positives. One common loop approximation is to "unroll" some small number of times. For example, unrolling $s^*$ two times, we get:

$$\texttt{skip} \ \square \ s \ \square \ (s; s)$$

In the context of other analyses, unrolling is often a reasonable approximation. However, it is an under-approximation, and therefore can lead to false positives in universal reachability analysis.

$$s^* \sqsupseteq \texttt{skip} \ \square \ s \ \square \ (s; s)$$

Intuitively, the reason unrolling a finite number of iterations can lead to false positives is that the body of the loop may contain assignment statements that update the store on every iteration along with conditional predicates that may only be true on certain

iterations and the unrolled loop explicitly executes some finite number of iterations, starting with the first. However, we can use the `havoctargets`($s$) statement to replace all store elements updated by the loop body with arbitrary values, obscuring which iteration is occurring. Unrolling the body once, and inserting `havoctargets`($s$), we have:

$$\texttt{skip} \ \square \ (\texttt{havoctargets}(s) \ ; \ s) \sqsupseteq s^*$$

We can also unroll more than once:

$$\left(\begin{array}{ll} & \texttt{skip} \\ \square & \texttt{havoctargets}(s); s \\ \square & \texttt{havoctargets}(s); s; \\ & \texttt{havoctargets}(s); s \end{array}\right) \sqsupseteq s^*$$

but this does not buy us anything. The weakest precondition of the loop unrolled twice is the same as the weakest precondition when unrolled only once, due to the havoc statements.

One disadvantage to the use of havoc statements is that they often obscure too much information, leading to false negatives. Ideally we would use a loop invariant, $I$, to describe what condition is true on every loop iteration:

$$\begin{array}{ll} & \texttt{assume}(I) \\ \square & \\ & \texttt{havoctargets}(s); \\ & \texttt{assume}(I); \\ & s; \\ & \texttt{assume}(I) \end{array}$$

If the loop invariant $I$ is too weak (*e.g.*, `true`), then we may miss violations of universal reachability, but we will get no false positives. If the loop invariant is too strong (*e.g.*, `false`), then one of the assertions may fail, which can lead to false reports of universal reachability violations. To perform verification in the presence of loops, rather than universal reachability analysis, precise loop invariants are necessary.

## 3.7 Universal Reachability Examples

This section illustrates the analysis described so far by applying it to a number of small examples containing common bugs, describing how each phase operates on each example. In this chapter, we will focus on how the formal presentation of the analysis operates. Later, in Sec. 7.5, we present the results of running our implementation on each of these examples. In the same chapter, we also present the results of running the implementation on a collection of large, widely-used open source programs.

For each example, we present a C implementation followed by a translation into $GC_{\leftrightarrows}$, from Sec. 2.1.2. In the $GC_{\leftrightarrows}$ version, we allow functions to have parameters and return values, for a closer correspondence with the original C code. We then describe how slicing would apply to the translated program, and show that a specific slice of interest is not terminable. In the examples that never raise exceptions, we use a simpler definition of the *slices* function in which the case for sequencing is symmetrical:

$$slices(C, s_1 \ ; \ s_2) = slices(C[\bullet \ ; \ s_2], s_1) \cup slices(C[s_1 \ ; \ \bullet], s_2)$$

For the last example, which does use exceptions, we use the definition of the *slices* function from Sec. 3.2.

We present each example both in C and $GC_{\leftrightarrows}$ for several reasons. First, the C version is often easier to understand. At the same time, the analysis works on $GC_{\leftrightarrows}$, and presenting the examples in these languages allows us to show derivations directly. Finally, we can run our implementation directly on the C versions, giving the results shown in Sec. 7.5.

### 3.7.1 Out-of-Memory Error Handling

As a first example, we have a simple skeleton of the error handler for out-of-memory situation (Fig. 3.3 and Fig. 3.4). The malloc function returns a null pointer if insufficient heap space remains to satisfy the request. We assume that the free function requires its argument to be non-null, so the $GC_{\leftrightarrows}$ version includes the appropriate assertion.

In this small example, the error is trivially visible. The program prints an error message if the allocation fails, but does not exit the program, and then attempts

```
void oom(int size) {
    char *buf = malloc(size);
    if(!buf) { printf("Failed to allocate memory"); }
    // Use buf
    free(buf);
}
```

**Figure 3.3:** Out-of-Memory Example (C)

```
buf := malloc(size);
  assume buf = 0;
  printf(...);
□
  assume buf ≠ 0;
assert buf ≠ 0;
free(buf);
```

**Figure 3.4:** Out-of-Memory Example ($GC_{\leftrightarrow}$)

to free the pointer that is guaranteed to be null. Similar patterns often appear in large programs where the allocation, error handler, and disposal functions are separated by many statements.

The function has two slices: one non-terminable and one terminable. The erroneous, non-terminable slice is as follows.

```
buf := malloc(size);
assume buf = 0;
printf(...);
assert buf ≠ 0;
free(buf);
```

We can show that its precondition is valid when using postconditions $N$ = `false`, $X$ = `false`, and $W$ = `true`. We leave out the printf call to simplify the derivation, since it does not contribute to the result.

$$
\begin{array}{l}
\quad wp( \\
\quad\quad \text{buf} := \text{malloc(size)}; \\
\quad\quad \texttt{assume}\ \text{buf} = 0; \\
\quad\quad \texttt{assert}\ \text{buf} \neq 0; \\
\quad\quad \text{free}\,(\text{buf}), \\
\quad \texttt{false}, \texttt{false}, \texttt{true}) \\
= \\
\quad wp( \\
\quad\quad \text{buf} := \text{malloc(size)}; \\
\quad\quad \texttt{assume}\ \text{buf} = 0; \\
\quad\quad \texttt{assert}\ \text{buf} \neq 0, \\
\quad \texttt{false}, \texttt{false}, \texttt{true}) \\
= \\
\quad wp( \\
\quad\quad \text{buf} := \text{malloc(size)}; \\
\quad\quad \texttt{assume}\ \text{buf} = 0, \\
\quad \text{buf} = 0, \texttt{false}, \texttt{true}) \\
= \\
\quad wp( \\
\quad\quad \text{buf} := \text{malloc(size)}; \\
\quad (\text{buf} = 0) \Rightarrow (\text{buf} = 0), \\
\quad \texttt{false}, \texttt{true}) \\
= \\
\quad \forall\ \text{buf'}.\ \text{buf'} = 0 \Rightarrow \text{buf'} = 0 \\
= \\
\quad \texttt{true}
\end{array}
$$

The terminable slice, on the other hand, has a non-valid precondition. The body of the slice consists of the following four statements.

$$
\begin{array}{l}
\text{buf} := \text{malloc(size)}; \\
\texttt{assume}\ \text{buf} \neq 0; \\
\texttt{assert}\ \text{buf} \neq 0;
\end{array}
$$

free(buf);

The terminability check for the slice is then as follows.

$$
\begin{array}{l}
\quad wp( \\
\quad\quad \text{buf} := \text{malloc(size)}; \\
\quad\quad \texttt{assume}\ \text{buf} \neq 0; \\
\quad\quad \texttt{assert}\ \text{buf} \neq 0; \\
\quad\quad \text{free(buf)}, \\
\quad \texttt{false}, \texttt{false}, \texttt{true}) \\
= \\
\quad wp( \\
\quad\quad \text{buf} := \text{malloc(size)}; \\
\quad\quad \texttt{assume}\ \text{buf} \neq 0; \\
\quad\quad \texttt{assert}\ \text{buf} \neq 0, \\
\quad \texttt{false}, \texttt{false}, \texttt{true}) \\
= \\
\quad wp( \\
\quad\quad \text{buf} := \text{malloc(size)}; \\
\quad\quad \texttt{assume}\ \text{buf} \neq 0, \\
\quad \text{buf} = 0, \texttt{false}, \texttt{true}) \\
= \\
\quad wp( \\
\quad\quad \text{buf} := \text{malloc(size)}; \\
\quad \text{buf} \neq 0 \Rightarrow \text{buf} = 0, \\
\quad \texttt{false}, \texttt{true}) \\
= \\
\quad \forall\ \text{buf'}.\ \text{buf'} = 0
\end{array}
$$

The resulting predicate is not valid, so this slice is terminable.

### 3.7.2 Double Free and Use After Free

Here we illustrate an example similar to the previous, but with important variations. In Fig. 3.5 we show a C program that allocates a buffer, executes some other statements, and then checks for an error condition. The $GC_{\not\rightsquigarrow}$ version appears in Fig. 3.6. If an error has occurred, it frees the buffer, but does not return from the function. Later, after various other statements have executed, it unconditionally frees the buffer.

In this case, we assume that the free function has a precondition that its argument is in the *allocated* state, and an effect of putting its argument into the *released* state. We model the state of the variable buf with the ghost variable $S_b$.

```
void dblfree(int size) {
    int error;
    char *buf = malloc(size);
    // Use buf and maybe set error
    if(error) { free(buf); }
    // Do other cleanup
    free(buf);
}
```

**Figure 3.5:** Double Free Example (C)

As we can see from an analysis of the slices of this program, if an error occurs during execution of the first collection of statements, then the program will attempt to release the buffer twice, and the second attempt will be guaranteed to fail. The failing slice is as follows.

buf := malloc(size);
$S_b$ := *allocated*;
`assume` error;
`assert` $S_b$ = *allocated*;
free(buf);
$S_b$ := *released*;

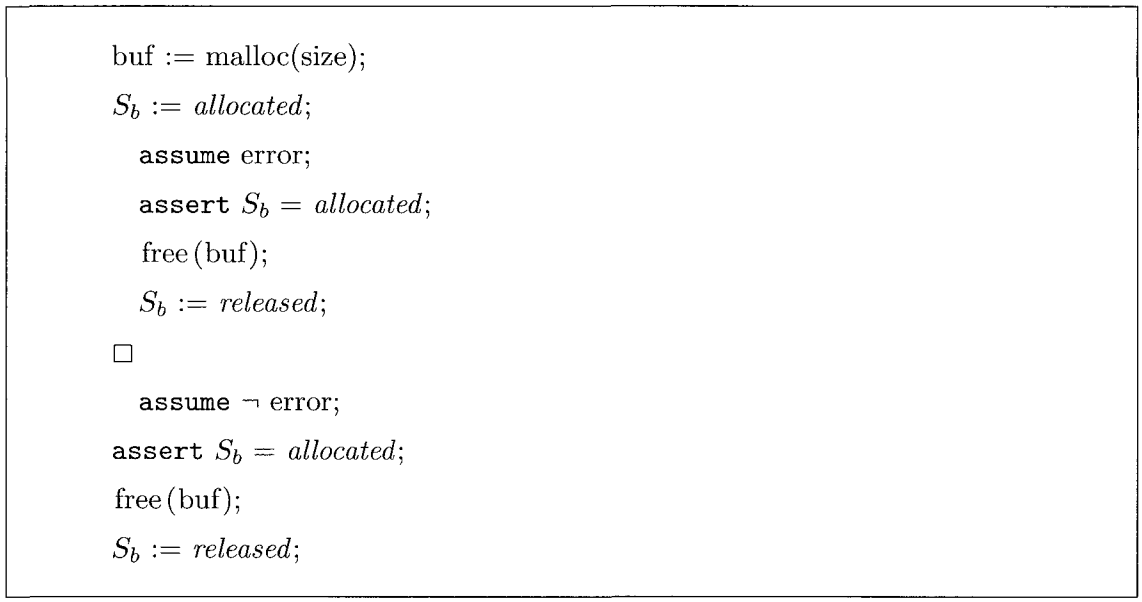


**Figure 3.6:** Double Free Example ($GC_{\leftrightarrow}$)

```
assert S_b = allocated;
free(buf);
S_b := released;
```

In this form, the inconsistency is obvious. We assign the value *released* to $S_b$, and then assert that it has the value *allocated*. Calculating the weakest precondition, we have:

```
  wp(
    buf := malloc(size);
    assume error;
    assert S_b = allocated;
    free(buf);
    S_b := released;
    assert S_b = allocated;
    free(buf);
    S_b := released,
  false, false, true)
=
  wp(
```

buf := malloc(size);
**assume** error;
**assert** $S_b = \mathit{allocated}$;
free(buf);
$S_b := \mathit{released}$;
**assert** $S_b = \mathit{allocated}$;
free(buf);
**false**, **false**, **true**)

=

$wp($
buf := malloc(size);
**assume** error;
**assert** $S_b = \mathit{allocated}$;
free(buf);
$S_b := \mathit{released}$;
**assert** $S_b = \mathit{allocated}$,
**false**, **false**, **true**)

=

$wp($
buf := malloc(size);
**assume** error;
**assert** $S_b = \mathit{allocated}$;
free(buf);
$S_b := \mathit{released}$,
$S_b \neq \mathit{allocated}$, **false**, **true**)

=

$wp($
buf := malloc(size);
**assume** error;
**assert** $S_b = \mathit{allocated}$;
free(buf),

$$
\begin{array}{l}
\quad \mathit{released} \neq \mathit{allocated}, \texttt{false}, \texttt{true}) \\
= \\
\quad wp( \\
\quad\quad \text{buf} := \text{malloc(size)}; \\
\quad\quad \texttt{assume error}; \\
\quad\quad \texttt{assert } S_b = \mathit{allocated}; \\
\quad \texttt{true}, \texttt{false}, \texttt{true}) \\
= \\
\quad wp( \\
\quad\quad \text{buf} := \text{malloc(size)}; \\
\quad\quad \texttt{assume error}; \\
\quad \texttt{true}, \texttt{false}, \texttt{true}) \\
= \\
\quad wp( \\
\quad\quad \text{buf} := \text{malloc(size)}; \\
\quad \texttt{true}, \texttt{false}, \texttt{true}) \\
= \\
\quad \texttt{true}
\end{array}
$$

And therefore the slice is non-terminable. We can also replace the second call to free with a dereference of the same pointer. If we assume that pointer dereferencing also requires its argument to be in the *allocated* state (or, alternatively, that it not be in the *released* state), we can also detect cases where a pointer is dereferenced after being released.

### 3.7.3 Binary Search

Recall the example of binary search from Ch. 1. We now go into detail about how universal reachability analysis applies to the program, repeated for easy reference in Fig. 3.7, along with an translation into $GC_{\leftrightarrows}$ in Fig. 3.8. In the translation to $GC_{\leftrightarrows}$, we assume that the function bs has a precondition that beg $<$ end and insert the appropriate assumptions and assertions.

If we assume that the precondition of bs is beg $<$ end, we can present an

```
int bs(int a[], int n, int beg, int end) {
    int mid = (beg + end) / 2;
    if(a[mid] == n) {
        return mid;
    } else if(a[mid] < n) {
        return bs(a, n, mid, end);
    } else if(a[mid] > n) {
        return bs(a, n, mid, beg);
    }
}
```

**Figure 3.7:** Binary Search Example (C)

expanded version of the slice presented in Ch. 1, in which the recursive call violates the precondition of bs.

```
assume beg < end;
mid := (beg + end)/2;
assume a[mid] ≠ n;
assume a[mid] ≥ n;
assume a[m] > n;
assert mid < beg;
ret := bs(a, n, mid, beg)
```

We can then derive the weakest precondition of this slice as follows:

```
wp(
  assume beg < end;
  mid := (beg + end)/2
  assume a[mid] ≠ n;
  assume a[mid] ≥ n;
  assume a[mid] > n;
  assert mid < beg;
  r := bs(a, n, mid, beg),
```

```
assume beg < end;
mid := (beg + end) / 2;
    assume a[mid] = n;
    ret := mid;
  □
    assume a[mid] ≠ n;
        assume a[mid] < n;
        assert mid < end;
        ret := bs(a, n, mid, end);
      □
        assume a[mid] ≥ n;
            assume a[mid] > n;
            assert mid < beg;
            ret := bs(a, n, mid, beg)
          □
            assume a[mid] ≤ n;
            skip
```

**Figure 3.8:** Binary Search Example ($GC_{\leftrightarrow}$)

```
  false, false, true)
=
  wp(
    mid := (beg + end)/2;
    assert beg < end;
    assume a[mid] ≠ n;
    assume a[mid] ≥ n;
    assume a[mid] > n;
    assert mid < beg,
  false, false, true)
=
```

$wp($

m := (beg + end)/2;

**assert** beg < end;

**assume** a[mid] $\neq$ n;

**assume** a[mid] $\geq$ n;

**assume** a[mid] > n,

mid $\geq$ beg, **false**, **true**)

=

$wp($

m := (beg + end)/2;

**assert** beg < end;

**assume** a[mid] $\neq$ n;

**assume** a[mid] $\geq$ n,

a[mid] > n $\Rightarrow$ mid $\geq$ beg, **false**, **true**)

=

$wp($

mid := (beg + end)/2;

**assert** beg < end;

**assume** a[mid] $\neq$ n,

a[mid] $\geq$ n $\Rightarrow$

a[mid] > n $\Rightarrow$ mid $\geq$ beg,

**false**, **true**)

=

$wp($

mid := (beg + end)/2;

**assert** beg < end,

a[mid] $\neq$ n $\Rightarrow$

a[mid] $\geq$ n $\Rightarrow$

a[mid] > n $\Rightarrow$ mid $\geq$ beg,

**false**, **true**)

=

$$
\begin{array}{l}
\quad wp( \\
\quad\quad \text{mid} := (\text{beg} + \text{end})/2, \\
\quad \text{beg} < \text{end} \Rightarrow \\
\quad\quad \text{a}[\text{mid}] \neq \text{n} \Rightarrow \\
\quad\quad\quad \text{a}[\text{mid}] \geq \text{n} \Rightarrow \\
\quad\quad\quad\quad \text{a}[\text{mid}] > \text{n} \Rightarrow \text{mid} \geq \text{beg}, \\
\quad \texttt{false}, \texttt{true}) \\
= \\
\quad \text{beg} < \text{end} \Rightarrow \\
\quad\quad \text{a}[(\text{beg} + \text{end})/2] \neq \text{n} \Rightarrow \\
\quad\quad\quad \text{a}[(\text{beg} + \text{end})/2] \geq \text{n} \Rightarrow \\
\quad\quad\quad\quad \text{a}[(\text{beg} + \text{end})/2] > \text{n} \Rightarrow ((\text{beg} + \text{end})/2) \geq \text{beg} \\
= \\
\quad \text{beg} < \text{end} \Rightarrow ((\text{beg} + \text{end})/2) \geq \text{beg} \\
= \\
\quad \texttt{true}
\end{array}
$$

Thus, we have shown that this slice is not terminable, in this case because it will violate the precondition of bs on the recursive call.

### 3.7.4 Doubly-Linked List

Next, we show a buggy implementation of a procedure to insert a new element into a doubly-linked list (Fig. 3.9 and Fig. 3.10), derived from an undergraduate programming assignment. In this case, the bug is less obvious than in the previous examples.

In the translation of the program into $GC_{\leftrightarrow}$ we represent the offset of a structure field "f" using the notation "O(f)". We also guard field dereferences with a precondition that requires the base expression to be non-zero. Since the variables new and list never change, we assert that they are non-null only once, at the beginning of the function body.

Now consider the case where the empty `else` branch of the first conditional executes. This means that the value of new→prev at that point is null. This implies

```
void List_insert ( List_ref  list , void *data) {
    Element_ref new = malloc(sizeof(struct Element));
    new→prev = list→curr;
    new→next = list→curr→next;
    if(new→prev) { new→prev→next = new; }
    if(new→next) {
        new→next→prev = new;
    } else {
        list →last = new;
    }
}
```

**Figure 3.9:** Doubly-Linked List Example (C)

that the value of list →curr is also null, which means that the line assigning new→next is guaranteed to fail with a null pointer dereference. Formally, this slice is as follows.

```
new := malloc(sizeof(Element));
assert new ≠ 0;
assert list ≠ 0;
M[new + O(prev)] := M[list + O(curr)];
assert(M[list + O(curr)] ≠ 0);
M[new + O(next)] := M[M[list + O(curr)] + O(next)];
assume M[new + O(prev)] = 0;
    assume M[new + O(next)] ≠ 0;
    assert M[new + O(next)] ≠ 0;
    M[M[new + O(next)] + O(prev)] := new
  □
    assume M[new + O(next)] = 0;
    M[list + O(last)] := new
```

We can now calculate the weakest precondition of this slice, with the standard postconditions for checking terminability.

```
new := malloc(sizeof(Element));
assert new ≠ 0;
assert list ≠ 0;
M[new + O(prev)] := M[list + O(curr)];
assert(M[list + O(curr)] ≠ 0);
M[new + O(next)] := M[M[list + O(curr)] + O(next)];
    assume M[new + O(prev)] ≠ 0;
    assert M[new + O(prev)] ≠ 0;
    M[M[new + O(prev)] + O(next)] := new
  □
    assume M[new + O(prev)] = 0
;
    assume M[new + O(next)] ≠ 0;
    assert M[new + O(next)] ≠ 0;
    M[M[new + O(next)] + O(prev)] := new
  □
    assume M[new + O(next)] = 0;
    M[list + O(last)] := new
```

**Figure 3.10:** Doubly-Linked List Example ($GC_{\leftrightarrow}$)

```
wp(
  new := malloc(sizeof(Element));
  assert new ≠ 0;
  assert list ≠ 0;
  M[new + O(prev)] := M[list + O(curr)];
  assert(M[list + O(curr)] ≠ 0);
  M[new + O(next)] := M[M[list + O(curr)] + O(next)];
  assume M[new + O(prev)] = 0;
      assume M[new + O(next)] ≠ 0;
      assert M[new + O(next)] ≠ 0;
```

M[M[new + O(next)] + O(prev)] := new

□

**assume** M[new + O(next)] = 0;

M[list + O(last)] := new,

**false**, **false**, **true**)

=

*wp*(

new := malloc(sizeof(Element));

**assert** new ≠ 0;

**assert** list ≠ 0;

M[new + O(prev)] := M[list + O(curr)];

**assert**(M[list + O(curr)] ≠ 0);

M[new + O(next)] := M[M[list + O(curr)] + O(next)];

**assume** M[new + O(prev)] = 0;

**assume** M[new + O(next)] ≠ 0;

**assert** M[new + O(next)] ≠ 0;

□

**assume** M[new + O(next)] = 0,

**false**, **false**, **true**)

=

*wp*(

new := malloc(sizeof(Element));

**assert** new ≠ 0;

**assert** list ≠ 0;

M[new + O(prev)] := M[list + O(curr)];

**assert**(M[list + O(curr)] ≠ 0);

M[new + O(next)] := M[M[list + O(curr)] + O(next)];

**assume** M[new + O(prev)] = 0,

M[new + O(next)] ≠ 0 ∧

M[new + O(next)] ≠ 0 ⇒ M[new + O(next)] = 0,

**false**, **true**)

$=$

$wp($

new := malloc(sizeof(Element));

`assert` new $\neq 0$;

`assert` list $\neq 0$;

M[new + O(prev)] := M[list + O(curr)];

`assert`(M[list + O(curr)] $\neq 0$);

M[new + O(next)] := M[M[list + O(curr)] + O(next)];

`assume` M[new + O(prev)] $= 0$,

`false`,

`false`, `true`)

$=$

$wp($

new := malloc(sizeof(Element));

`assert` new $\neq 0$;

`assert` list $\neq 0$;

M[new + O(prev)] := M[list + O(curr)];

`assert`(M[list + O(curr)] $\neq 0$);

M[new + O(next)] := M[M[list + O(curr)] + O(next)],

M[new + O(prev)] $\neq 0$,

`false`, `true`)

$=$

$wp($

new := malloc(sizeof(Element));

`assert` new $\neq 0$;

`assert` list $\neq 0$;

M[new + O(prev)] := M[list + O(curr)];

`assert`(M[list + O(curr)] $\neq 0$),

M[new + O(prev)] $\neq 0$,

`false`, `true`)

$=$

$wp($

new := malloc(sizeof(Element));

`assert` new $\neq 0$;

`assert` list $\neq 0$;

M[new + O(prev)] := M[list + O(curr)];

`assert`(M[list + O(curr)] $\neq 0$),

M[list + O(curr)] $\neq 0 \Rightarrow$

M[new + O(prev)] $\neq 0$,

`false`, `true`)

$=$

$wp($

new := malloc(sizeof(Element));

`assert` new $\neq 0$;

`assert` list $\neq 0$;

M[new + O(prev)] := M[list + O(curr)],

M[list + O(curr)] $\neq 0 \Rightarrow$

M[list + O(curr)] $\neq 0 \Rightarrow$

M[new + O(prev)] $\neq 0$,

`false`, `true`)

$=$

$wp($

new := malloc(sizeof(Element));

`assert` new $\neq 0$;

`assert` list $\neq 0$,

M[list + O(curr)] $\neq 0 \Rightarrow$

M[list + O(curr)] $\neq 0 \Rightarrow$

M[list + O(curr)] $\neq 0$,

`false`, `true`)

$=$

$wp($

new := malloc(sizeof(Element));

$$
\begin{array}{l}
\quad \texttt{assert new} \neq 0, \\
\quad \texttt{true}, \texttt{false}, \texttt{true}) \\
= \\
\quad wp( \\
\quad\quad \text{new} := \text{malloc(sizeof(Element))}, \\
\quad \texttt{true}, \texttt{false}, \texttt{true}) \\
= \\
\quad \texttt{true}
\end{array}
$$

Therefore, this slice is non-terminable.

### 3.7.5 Exception Handling

To illustrate exception handling in $GC_{\leftrightarrows}$, we present an example in Fig. 3.11 that attempts to acquire an instance of some resource, where the acquire function may throw an exception if something goes wrong, as may the statement s that executes once the resources have been allocated. As in the memory allocation examples, we use the ghost variable $S_x$ to represent the current state of the resource that x refers to.

```
    x := create();
    S_x := initial;
    acquire(x);
    S_x := acquired;
    s
!
    assert S_x = acquired;
    release(x);
    S_x := released;
```

**Figure 3.11:** Exception Handling Example ($GC_{\leftrightarrows}$)

In this example, the exception handler was written to handle exceptions that may occur during the execution of s, but not the exceptions that may occur during the resource acquisition phase. If the exception handler executes as a result of one of this

earlier exception, it will attempt to release a resource that has not been acquired.

Because we assume that the acquire function may throw an exception, and because the rest of its specification is already in place, in the form of explicit assertions and assignments involving the ghost state variable, we can desugar the acquire function into `skip` $\square$ `raise`, resulting in the following program:

x := create();
$S_x$ := *initial*;
(`skip` $\square$ `raise`);
$S_x$ := *acquired*;
s
!
`assert` $S_x$ = *acquired*;
release (x);
$S_x$ := *released*;

The slices of this program are a bit more complex than in the previous examples, because of the exception handling construct we must insert after the first statement of each instance of the sequencing operator. One of the problematic slices is as follows:

x := create() ! $\bot$;
$S_x$ := *initial* ! $\bot$;
`raise`;
$S_x$ := *acquired*;
s
!
`assert` $S_x$ = *acquired*;
release (x);
$S_x$ := *released*;

Note the additional exception handlers containing $\bot$ after the first two statements, inserted by the *slices* function. Recall that the we use $\bot$ to refer to a divergence, such as the specific statement `assume(false)`. To calculate the terminability of this slice, we first need to calculate the weakest precondition of the exception handler, given postconditions $N =$ `false`, $X =$ `false`, $W =$ `true`.

$$
\begin{array}{l}
\quad wp( \\
\quad\quad \texttt{assert}\ S_x = \mathit{acquired}; \\
\quad\quad \text{release}\,(\text{x}); \\
\quad\quad S_x := \mathit{released}, \\
\quad \texttt{false}, \texttt{false}, \texttt{true}) \\
= \\
\quad wp( \\
\quad\quad \texttt{assert}\ S_x = \mathit{acquired}; \\
\quad\quad \text{release}\,(\text{x}), \\
\quad \texttt{false}, \texttt{false}, \texttt{true}) \\
= \\
\quad wp( \\
\quad\quad \texttt{assert}\ S_x = \mathit{acquired}, \\
\quad \texttt{false}, \texttt{false}, \texttt{true}) \\
= \\
\quad S_x \neq \mathit{acquired}
\end{array}
$$

We then use the result as the exceptional postcondition when calculating the weakest precondition of the earlier statements. Let $P_s = wp(s, \texttt{false}, \texttt{false}, \texttt{true})$.

$$
\begin{array}{l}
\quad wp( \\
\quad\quad \text{x} := \text{create}() \ !\ \bot; \\
\quad\quad S_x := \mathit{initial} \ !\ \bot; \\
\quad\quad \texttt{raise}; \\
\quad\quad S_x := \mathit{acquired}; \\
\quad\quad \text{s}, \\
\quad \texttt{false}, S_x \neq \mathit{acquired}, \texttt{true}) \\
= \\
\quad wp( \\
\quad\quad \text{x} := \text{create}() \ !\ \bot; \\
\quad\quad S_x := \mathit{initial} \ !\ \bot; \\
\quad\quad \texttt{raise}; \\
\quad\quad S_x := \mathit{acquired},
\end{array}
$$

$P_s, S_x \neq \mathit{acquired}, \texttt{true})$

$=$

$wp($

x := create() ! $\bot$;

$S_x := \mathit{initial}$ ! $\bot$;

`raise`,

$P_s[S_x := \mathit{acquired}], S_x \neq \mathit{acquired}, \texttt{true})$

$=$

$wp($

x := create() ! $\bot$:

$S_x := \mathit{initial}$ ! $\bot$,

$S_x \neq \mathit{acquired}, S_x \neq \mathit{acquired}, \texttt{true})$

$=$

$wp($

x := create() ! $\bot$;

$S_x := \mathit{initial}$,

$S_x \neq \mathit{acquired}, \texttt{true}, \texttt{true})$

$=$

$wp($

x := create() ! $\bot$,

$\mathit{initial} \neq \mathit{acquired}, \texttt{true}, \texttt{true})$

$=$

$wp($

x := create(),

$\mathit{initial} \neq \mathit{acquired}, \texttt{true}, \texttt{true})$

$=$

$\mathit{initial} \neq \mathit{acquired}$

$=$

`true`

So we can see that if the acquire function raises an exception, an assertion failure is guaranteed.

# Chapter 4

# Unstructured Programs

The previous chapter described universal reachability analysis on an idealized, structured language. The approach taken thus far helps clarify the concepts involved, but does not lend itself to a practical implementation for the imperative programming languages in widespread use.

This chapter now describes some modifications necessary for practical and scalable support of languages such as C. First we discuss control flow as it occurs in partially structured languages. We adapt the weakest precondition computation using a variant of a known technique, and introduce a slicing algorithm for instruction graphs. We then describe a previously known technique for improving efficiency by reducing duplication in verification (or failure) conditions. This technique is not essential for the correctness of our analysis, but makes it practical for us to analyze programs consisting of millions of lines of source code, and we present measurements of its impact in Sec. 7.9.3.

## 4.1 Unstructured Language Syntax

Although our analysis so far has focused on an inductive syntax, it turns out that the approach described in Ch. 3 can be adapted fairly easily to programs represented as control-flow graphs. In this section, we modify the syntax of the $GC_{\leftrightarrow}$ language from Sec. 2.1.2 to use arbitrary jumps or gotos in the place of sequencing, loops, and exceptions.

An unstructured program is a graph of instructions augmented with several distinguished vertices. We will use the metavariable $G$ to represent a graph of the form $(V, E)$, where $V$ is a set of vertices and $E$ is a set of edges. We use the metavariable $B$ to represent a function body, which takes the form $(v_S, v_N, v_X, G)$. The distinguished vertex $v_S$ is the entry point for the function, $v_N$ is the normal termination point, and $v_X$ is the exceptional termination point. Vertices in $V$ are labeled with atomic instructions, $i$, from the syntax of $GC_{\leftrightarrow}$, and each pair $(u, v)$ in $E$ represents an edge from $u$ to $v$ in the direction of program control flow. We will use the notation $E(G)$ and $E(B)$ to represent the edges of a graph or function body, respectively. Similarly, we will use the notation $V(G)$ and $V(B)$ to extract the vertices of a graph or body.

For formal discussions, we will treat unstructured programs as graphs, not syntax. However, for presentation of examples, we do provide a linear, textual syntax for these graphs, $IR_{\leftrightarrow}$, described in Fig. 4.1. All syntactic productions undefined in Fig. 4.1 are taken from the $GC_{\leftrightarrow}$ syntax of Fig. 2.1. The one exception is that we use `skip` as syntactic sugar for `assume(true)` so that it can be used as an instruction.

| | | | | |
|---|---|---|---|---|
| *Program* : | $p$ | ::= | $d^*$ | |
| *Function* : | $d$ | ::= | `fun` $f()$ $P$ { $b^*$ } $N$ $X$ | |
| *Block* : | $b$ | ::= | $l$ : $i^*$; $r$ | |
| *BlockEnd* : | $r$ | ::= | `return` | normal return |
| | | \| | `raise` | exceptional return |
| | | \| | `goto` $l^*$ | goto |
| *Label* : | $l$ | ::= | ... | |

**Figure 4.1:** Syntax for $IR_{\leftrightarrow}$

In $IR_{\leftrightarrow}$, programs are presented in terms of basic blocks, each beginning with a label ($l$ :) and ending with a special block end instruction ($r$). The labels and block end instructions describe the control-flow structure and do not explicitly exist in the graph format. In the translation to graph format, consecutive instructions in blocks are connected with a single forward edge. The notation `goto` $l^*$ describes an edge between the last instruction before the `goto` and the first instruction of each of the blocks listed. Non-deterministic choice is encoded using multiple outgoing edges. The

`return` instruction is translated into an edge to the normal termination vertex of the function ($v_N$). The `raise` instruction is translated into an edge to either the first instruction of the enclosing exception handler or the special exceptional termination vertex for the function ($v_X$).

The block-ending constructs, `goto`, `return`, and `raise` are not given a formal semantics, as they do not appear in the graph structure used for formal analysis. Intuitively, `return` indicates that the associated vertex should be analyzed using the normal postcondition, and `raise` indicates that the associated vertex should be analyzed using the exceptional postcondition.

A function body is considered well-formed if its graph has no vertices with an out-degree greater than two. The compilation relation described in the next section maintains this invariant, and larger numbers of branches can be encoded using intermediate vertices labeled with `skip`.

## 4.2 Translating Structured to Unstructured Programs

To translate a statement into an instruction graph, we use the compilation judgement $s \blacktriangleright B$ from Fig. 4.4, which makes use of an the auxiliary judgement $v_N, v_X \vdash s \triangleright (v_S, G)$ from Fig. 4.2 and Fig. 4.3. Given a normal termination vertex $v_N$, an exceptional termination vertex $v_X$, and a statement $s$, the judgement $v_N, v_X \vdash s \triangleright (v_S, G)$ produces a (sub-) graph $G$ equivalent to the statement, paired with an entry vertex $v_S$.

Compilation makes use of two special graph operators. The first, $\uplus$, combines two graphs by performing standard set union on the vertex and edge sets.

$$(V_1, E_1) \uplus (V_2, E_2) = (V_1 \cup V_2, E_1 \cup E_2)$$

The second, $\between$, indicates that two graphs are disjoint in their labeled vertices, and in the starting vertices of their edges. The graphs may include edges that terminate at shared vertices.

$$\begin{aligned}(V_1, E_1) \between (V_2, E_2) \quad &= \quad \{v \mid (v, i_1) \in V_1 \wedge (v, i_2) \in V_2\} = \emptyset \\ &\wedge \quad \{u \mid (u, v_1) \in E_1 \wedge (u, v_2) \in E_2\} = \emptyset\end{aligned}$$

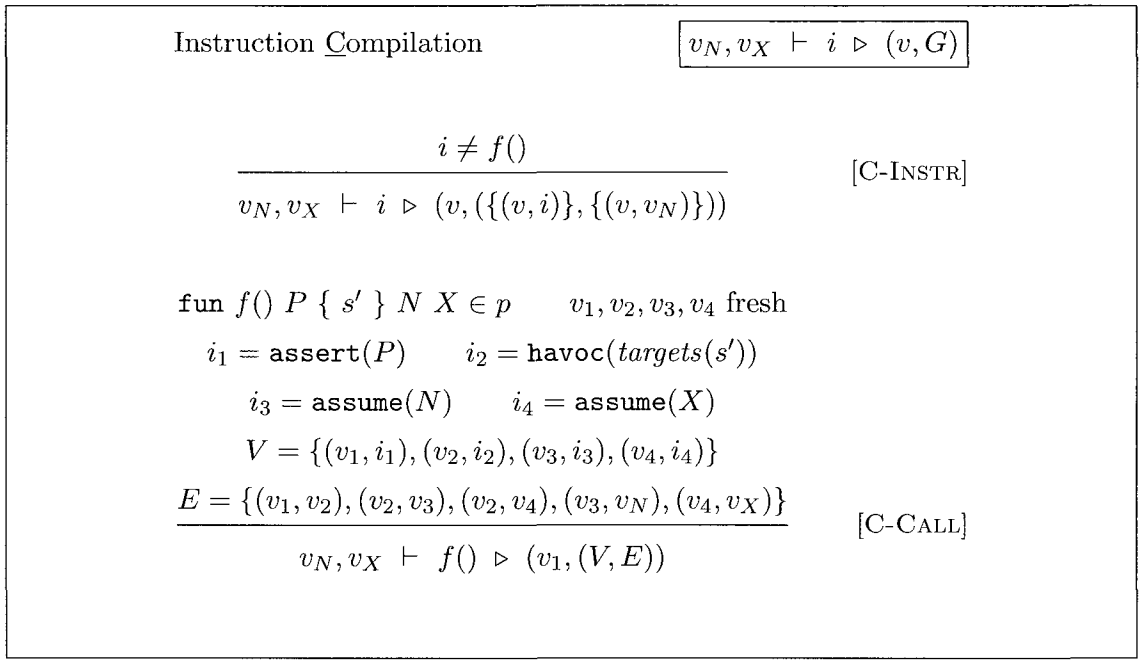


**Figure 4.2:** Compiling $GC_{\leftrightarrow}$ Instructions to $IR_{\leftrightarrow}$

To compile a bare instruction $i$ other than a call instruction, we use the rule [C-INSTR]. This yields a graph with a single fresh vertex $v$ labeled with $i$, and an edge from $v$ to the given normal termination vertex $v_N$. The entry vertex of the resulting body is $v$.

To compile a call instruction, we must take into account the possibility that the function can raise an exception. We handle this in [C-CALL] by compiling each call instruction $f()$ by looking up its definition, $\texttt{fun}\ f()\ P\ \{\ s'\ \}\ N\ X$, and creating four new instructions equivalent to the expanded statement used in the weakest precondition function. The first instruction asserts the precondition $P$, and continues directly to the second, which applies the havoc operation to all targets of $s'$. This second instruction has edges to two following instructions: one which assumes the normal postcondition and continues on to the normal termination vertex, and one which assumes the exceptional postcondition and continues on to the exceptional termination vertex. We perform this expansion with havoc because we use unstructured graphs only for analysis. Unstructured graphs have no operational semantics, so we have no need for the actual body of the target function.

For a choice statement, using [C-CHOICE], we first create a fresh branch vertex

Statement Compilation $\boxed{v_N, v_X \vdash s \triangleright (v, G)}$

$$\frac{v_N, v_X \vdash s_1 \triangleright (v_1, G_1) \qquad v_N, v_X \vdash s_2 \triangleright (v_2, G_2) \qquad v \text{ fresh} \qquad G_1 \between G_2}{v_N, v_X \vdash s_1 \;\square\; s_2 \triangleright (v, (\{(v, \texttt{skip})\}, \{(v, v_1), (v, v_2)\}) \uplus G_1 \uplus G_2)} \quad \text{[C-Choice]}$$

$$\frac{v \text{ fresh}}{v_N, v_X \vdash \texttt{skip} \triangleright (v, (\{(v, \texttt{skip})\}, \{(v, v_N)\})} \quad \text{[C-Skip]}$$

$$\frac{v \text{ fresh}}{v_N, v_X \vdash \texttt{raise} \triangleright ((v, (\{(v, \texttt{skip})\}, \{(v, v_X)\})} \quad \text{[C-Raise]}$$

$$\frac{v_2, v_X \vdash s_1 \triangleright (v_1, G_1) \qquad v_N, v_X \vdash s_2 \triangleright (v_2, G_2) \qquad G_1 \between G_2}{v_N, v_X \vdash s_1 \;;\; s_2 \triangleright (v_1, G_1 \uplus G_2)} \quad \text{[C-Seq]}$$

$$\frac{v_N, v_2 \vdash s_1 \triangleright (v_1, G_1) \qquad v_N, v_X \vdash s_2 \triangleright (v_2, G_2) \qquad G_1 \between G_2}{v_N, v_X \vdash s_1 \,!\, s_2 \triangleright (v_1, G_1 \uplus G_2)} \quad \text{[C-Catch]}$$

$$\frac{v, v_X \vdash s_1 \triangleright (v_1, G_1) \qquad v \text{ fresh}}{v_N, v_X \vdash s_1^* \triangleright (v, (\{(v, \texttt{skip})\}, \{(v, v_N), (v, v_1)\}) \uplus G_1)} \quad \text{[C-Loop]}$$

**Figure 4.3:** Compiling $GC_{\leftrightarrow}$ Statements to $IR_{\leftrightarrow}$

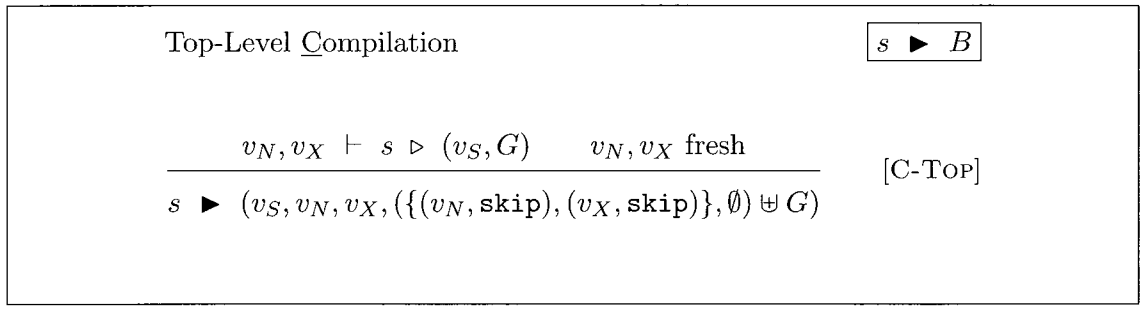


**Figure 4.4:** Compiling $GC_{\leftrightarrow}$ Programs to $IR_{\leftrightarrow}$

$v$ that is not in $G_1$ or $G_2$, with the associated instruction `skip`, which becomes the entry point of the resulting body. We then compile each of the branches with the given normal termination and exceptional termination vertices and combine the results.

To compile `skip` using [C-Skip], we create a body with a single vertex $v$ labeled with `skip`. We then add an edge from $v$ to the normal termination vertex $v_N$. The case for `raise` ([C-Raise]) is symmetrical, using the exceptional termination vertex as the target vertex of the new edge.

To compile a sequencing statement $s_1$ ; $s_2$ using [C-Seq], we compile $s_2$ with normal and exceptional termination vertices $v_N$ and $v_X$, resulting in a body $(v_2, G_2)$. We then compile $s_1$ with normal termination vertex $v_2$ and exceptional termination vertex $v_X$, resulting in a body $(v_1, G_1)$. The final result uses $v_1$ as its entry point, and combines graphs $G_1$ and $G_2$.

We compile exception handlers in a symmetrical way, according to [C-Catch]. To translate $s_1$ ! $s_2$, we compile $s_2$ with normal and exceptional termination vertices $v_N$ and $v_X$, resulting in a body $(v_2, G_2)$. We then compile $s_1$ with normal termination vertex $v_N$ and exceptional termination vertex $v_2$, resulting in a body $(v_1, G_1)$. As with sequencing, the final result uses $v_1$ as its entry point, and combines graphs $G_1$ and $G_2$.

To compile a loop $s^*$ using [C-Loop] we introduce a fresh vertex $v$, labeled with `skip`, and compile the body of the loop with $v$ as the normal termination vertex, resulting in a body $(v_1, G_1)$. We then introduce an edge from $v$ to $v_1$ (the "back" edge of the loop), as well as an edge from $v$ to $v_N$ (the "skip" edge).

Finally, to compile a top-level statement to a body using [C-Top], we compile the statement $s$ to $(v, G)$ in the context of two fresh vertices, $v_N$ and $v_X$, representing the normal and exceptional termination points, respectively. We then extend $G$ by

labeling $v_N$ and $v_X$ with the no-op instruction `skip`.

## 4.3 Weakest Preconditions

Weakest precondition analysis has traditionally been applied to structured programs. Dijkstra introduced the weakest precondition transformer, and was a vocal opponent of the use of unstructured control flow [35].

However, it turns out that the approach of weakest precondition analysis also applies naturally to unstructured control-flow graphs, as described by Barnett *et al.* [11], as well as Grigore *et al.* [52]. We present a set of inference rules for weakest preconditions of unstructured programs in Fig. 4.5. While Barnett *et al.* present their analysis as the result of calculating the solution to a system of equations, as do Grigore *et al.*, we define our result as the least relation that satisfies the given inductive definition. The two approaches yield the same results, but ours simplifies the proof of Thm. 6.

Our treatment takes exceptions into account (which the works mentioned previously omit, since they are less essential in unstructured programs), as well as the postcondition for going wrong that we have used for universal reachability analysis by using the previously-defined $wp(i, N, X, W)$ function to analyze instructions. Note, however, that these two postconditions are constant throughout the body of a given function. Only the $N$ predicate propagates through the graph.

We can show that the weakest precondition of a structured program is logically equivalent to its weakest precondition after compilation. For simplicity we assume that loops have been approximated either before compilation, using one of the techniques described in Sec. 3.6, or after compilation, using the technique of Sec. 4.3.1.

**Theorem 5.** *For any loop-free statement $s$, if $s \blacktriangleright (v_S, v_N, v_X, G)$ and $v_N, v_X, G \vdash uwp(v_S, N, X, W) = P$, then $wp(s, N, X, W) \Leftrightarrow P$.*

*Proof.* By induction on the derivation of $s \blacktriangleright (v_S, v_N, v_X, G)$. The full proof appears in Appendix E. □

Because the graph is acyclic, we can calculate the relation defined by the weakest precondition judgement bottom-up, starting from the vertices $v_N$ and $v_X$.

Unstructured Weakest Preconditions $\boxed{v_N, v_X, G \vdash uwp(v, N, X, W) = P}$

$$\frac{}{v_N, v_X, G \vdash uwp(v_N, N, X, W) = N} \quad \text{[U-XTerm]}$$

$$\frac{}{v_N, v_X, G \vdash uwp(v_X, N, X, W) = X} \quad \text{[U-NTerm]}$$

$$\frac{\begin{array}{c} v \text{ has outdegree 1} \\ (v, v') \in G \qquad (v, i) \in G \\ v_N, v_X, G \vdash uwp(v', N, X, W) = P' \\ wp(i, P', X, W) = P \end{array}}{v_N, v_X, G \vdash uwp(v, N, X, W) = P} \quad \text{[U-Instr]}$$

$$\frac{\begin{array}{c} v \text{ has outdegree 2} \\ (v, v_1) \in G \qquad (v, v_2) \in G \\ v_1 \neq v_2 \qquad (v, i) \in G \\ v_N, v_X, G \vdash uwp(v_1, N, X, W) = P_1 \\ v_N, v_X, G \vdash uwp(v_2, N, X, W) = P_2 \\ wp(i, P_1 \wedge P_2, X, W) = P \end{array}}{v_N, v_X, G \vdash uwp(v, N, X, W) = P} \quad \text{[U-Choice]}$$

**Figure 4.5:** Weakest Preconditions of $IR_{\leftrightarrow}$ Graphs

If the program of interest originated as a structured program, we can use the approximations described in Sec. 3.6 to remove loops. However, if the program is unstructured from the start, we need a way to create acyclic approximations of cyclic control flow graphs.

### 4.3.1 Translation to Loop-Free Programs

The syntax from Fig. 4.1 allows arbitrary control flow, including standard loops, exceptions, and less structured constructs. However, just as structured loops present difficulty to the analysis of structured programs, loops in the control-flow graphs of unstructured programs must be translated to some approximate closed form before weakest precondition analysis, unless we have the ability to reason directly about fixpoints.

In Sec. 3.6 we discussed the treatment of loops in universal reachability analysis, with the conclusion that analyzing a single, unrestricted iteration of each loop was sufficient for checking universal reachability. To consider the body of a loop as a completely arbitrary iteration, we described the insertion of a havoc statement at the beginning of the loop body, which obliterates knowledge of the values of any variable written within the loop body.

We follow a similar approach, taken from Barnett *et al.* [11], to remove loops from unstructured programs. First, we use standard techniques to transform the control flow graph into a reducible graph [3]. Then, for each back edge in the graph of a function body, we insert a havoc instruction at the target of the back edge (*i.e.*, the head of the loop) and remove the back edge itself. The havoc instruction obliterates the value of any variable written on any path from the target of the back edge to the source of the back edge. Formally, for each back edge, $(u, v)$, we calculate the following set of targets.

$$\begin{aligned} \mathit{utargets}(u, v, G) = \{\ x \quad \mid \quad & (v, v') \in E(G)^+ \wedge \\ & (v', u) \in E(G)^+ \wedge \\ & (v', i) \in V(G) \wedge \\ & x \in \mathit{targets}(i)\ \} \end{aligned}$$

We use $E(G)^+$ to describe the transitive closure of the set of edges in $G$. We then insert a new instruction, $\texttt{havoc}(\mathit{utargets}(u, v))$, with all of the incoming edges originally

destined for $v$ and with an outgoing edge to $v$.

## 4.4 Slicing Unstructured Programs

Slicing is a critical component of the algorithm for checking universal reachability. However, the function to calculate the slices of a structured programs does not translate directly to unstructured programs. Fortunately, the intuition behind the purpose of slicing does have a natural counterpart for unstructured programs. We present the slicing algorithm for unstructured programs below, and prove that it is equivalent to the slicing function for structured programs.

The key intuition behind slicing is that each slice should represent a variant of the original program in which only one side of a given choice statement is possible. To force the execution of a given choice branch, it is also necessary to force any enclosing choice vertices to take the branch that leads to the selected choice statement.

Another way of describing this intuition is that a slice forces the execution of a program to traverse an edge in the program graph that would be optional in the original program. Therefore, the unstructured slicing algorithm consists of two phases. The first is responsible for identifying the graph edges that correspond to slices. The second is responsible for creating a new graph for each of the chosen slice edges in which every path traverses the slice edge.

We want to generate one slice for each conditional statement in the program. Therefore, every edge leading out of a vertex with an out-degree greater than one is a slice edge. The function *uslice* in Fig. 4.6 calculates the entire slice associate with a slice edge, by performing a reachability traversal from the target vertex of the slice edge, performing a reverse reachability traversal from the source vertex of the slice edge, and connecting the graphs resulting from these traversals using the slice edge itself, omitting any edges parallel to the slice edge.

The reverse reachability traversal is necessary to ensure that the slice forces any earlier branches that lead to the slice edge. A consequence of using reachability, however, is that the slicing algorithm will produce the correct result only on an acyclic graph, and therefore must be performed after loop approximation.

The *uslices* function, also in Fig. 4.6, takes a graph as input, identifies all slice

To generate one slice of a graph $G$, given an edge:

$$
\begin{aligned}
\mathit{uslice} &: \mathit{Graph} \times \mathit{Edge} \to \mathit{Graph} \\
\mathit{uslice}((V,E),(u,v)) &= (V,E') \\
\text{where } E' &= \{(u',v') \mid (u',v') \in E \wedge (v',u) \in E^+\} \\
&\cup \{(u',v') \mid (u',v') \in E \wedge (v,u') \in E^+\} \\
&\cup \{(u,v)\}
\end{aligned}
$$

To generate all slices of a graph $G$:

$$
\begin{aligned}
\mathit{uslices} &: \mathit{Graph} \to 2^{\mathit{Graph}} \\
\mathit{uslices}(G) &= G \cup \{\mathit{uslice}(G,(u,v)) \mid (u,v) \in E(G) \wedge \\
&\qquad (u,v') \in E(G) \wedge \\
&\qquad v \neq v' \wedge \\
&\qquad (u, \texttt{skip}) \in V(G)\}
\end{aligned}
$$

**Figure 4.6:** Slicing $IR_{\leftrightarrow}$ Graphs

edges in the graph, and uses *uslice* to extract the slice corresponding to each slice edge. Because compilation for function calls introduces multiple outgoing edges, but slicing on structured programs does not generate multiple slices for function calls, we restrict slicing of unstructured programs to consider multiple outgoing edges from `skip` vertices only.

Given these definitions, we can show that compilation of a loop-free and exception-free program followed by unstructured slicing is equivalent to structured slicing followed by compilation.

**Theorem 6.** *The following two statements are true of any loop-free and exception-free statement $s$:*

- *If $s' \in \mathit{slices}(\bullet, s)$ and $s \blacktriangleright (v_S, v_N, v_X, G)$ then there exists a $G' \in \mathit{uslices}(G)$ such that $v_N, v_X, G' \vdash \mathit{uwp}(v_S, N, X, W) = P$ and $\mathit{wp}(s', N, X, W) \Leftrightarrow P$.*
- *If $s \blacktriangleright (v_S, v_N, v_X, G)$ and $G' \in \mathit{uslices}(G)$ and $v_N, v_X, G' \vdash \mathit{uwp}(v_S, N, X, W) = P$ then there exists an $s' \in \mathit{slices}(\bullet, s)$ such that $\mathit{wp}(s', N, X, W) \Leftrightarrow P$.*

*Proof.* By induction on the structure of $s$. The full proof appears in Appendix F. □

## 4.5 Efficiency

It is well-known that the verification conditions (and failure conditions) generated from weakest precondition analysis can be exponential in the size of the original program, without special attention to explicitly sharing duplicated subterms [48, 11, 52]. The blowup in size occurs in the analysis of two language constructs. When analyzing assignments, every instance of the assigned variable in the postcondition is replaced by the expression on the right, and the assigned variable may occur many times. Similarly, when analyzing choice statements, the current postcondition is duplicated and passed through the weakest precondition transformer for both statements.

Consequently, various researchers have proposed more efficient weakest precondition calculation strategies. Flanagan and Saxe [48] provided a technique for efficient weakest precondition generation from structured programs with exceptions. Later work extended this work to deal with unstructured programs [11] and strongest postconditions [52]. The latter also provided some optimality bounds, in terms of the number additional temporary incarnations created and the number of "copy" operations needed.

The primary technique that all three works present to improve the efficiency of weakest precondition calculation is the creation of a *passive* program. In a passive program, each variable can have multiple incarnations, and assignments are replaced by assumptions that relate the value of one incarnation to earlier incarnations or other variables.

Grigore *et al.* present an algorithm for creating passive (unstructured) programs which is guaranteed to be optimal in terms of the number of copy statements it inserts [52]. We use the passivation algorithm of Grigore *et al.*, described next, in our implementation.

### 4.5.1 Passivation

All three works on efficient weakest precondition computation describe a transformation they call *passivation* or *passification* on programs before weakest precondition analysis. In each case, this transformation introduces multiple incarnations of each variable and replaces assignment statements with assumption statements relating the different incarnations. To facilitate translation of programs into passive form, we gen-

eralize the form of instructions somewhat, allowing the additional notation $\{\imath_1, \ldots, \imath_n\}$ to denote a set of $n$ instructions, and allow instruction sets to label a single vertex. In passive programs, the order of evaluation is irrelevant, so we can consider all of the instructions in an instruction set to be evaluated in parallel.

Flanagan and Saxe describe the process of creating passive programs in a structured language with exceptions. Barnett elides many of the details of the process on unstructured programs, but Grigore *et al.* go into detail and gives an analysis of complexity bounds of an imperative algorithm. We present the algorithm of Grigore *et al.* in the form of several pure functions: *iswrite*, $R$, $W$, *copy*, and the top-level function *passivate* in Fig. 4.7.

The passivation algorithm we present applies only to acyclic instruction graphs. Therefore, it is necessary to break any loops in the source program before passivation, using a technique such as that from Sec. 4.3.1. Slicing can occur either before or after passivation. Our implementation performs passivation first, to avoid duplicated work.

The mutually recursive functions $R$ and $W$ generate variable incarnation mappings, indicating which version of a variable $x$ is read or written by the instruction associated with vertex $v$. The incarnation of $x$ read at vertex $v$ is the maximum incarnation written by any of the predecessors of $v$. As incarnations are natural numbers, the maximum of the empty set is zero.

If the instruction at $v$ writes to variable $x$, then the incarnation written by $v$ is one greater than the incarnation read at $v$. Otherwise, it is the same as the incarnation read at $v$. Computation of $W$ uses the auxiliary function $\mathit{iswrite}(v, x, G)$, which returns 1 if the instruction on the vertex $v$ in graph $G$ writes to the variable $x$, and 0 otherwise. If $R$ and $W$ are computed using memoization, the complexity results given by Grigore *et al.* apply.

The results of the $R$ and $W$ functions are functions themselves, both with type $\mathit{Variable} \to \mathbb{N}$. We lift these functions to apply to expressions and predicates by applying to all variables within. We then apply them in the definition of the function *update*, which updates an instruction to passive form, based on the appropriate read and write incarnations for its location in the graph.

Whenever an edge $(u, v)$ exists in $G$, and $R(v, G)$ is not equal to $W(u, G)$, we

$$
\begin{aligned}
\mathit{iswrite} &: \mathit{Vertex} \times \mathit{Graph} \times \mathit{Variable} \to \mathbb{N} \\
\mathit{iswrite}(v, G, x) &= \begin{cases} 1 & \text{if } (v, \imath) \in V(G) \text{ and } x \in \mathit{targets}(\imath) \\ 0 & \text{otherwise} \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
R, W &: \mathit{Vertex} \times \mathit{Graph} \to \mathit{Variable} \to \mathbb{N} \\
R(v, G) &= \lambda x.\ \max\ \{ W(u, G)(x) \mid (u, v) \in E(G) \} \\
W(v, G) &= \lambda x.\ \mathit{iswrite}(v, G, x) + R(v, G)(x)
\end{aligned}
$$

$$
\begin{aligned}
\mathit{update} &: \mathit{Vertex} \times \mathit{Instruction} \times \mathit{Graph} \to \mathit{Statement} \\
\mathit{update}(\texttt{assert}(P), v, G) &= \texttt{assert}(R(v, G)(P)) \\
\mathit{update}(\texttt{assume}(P), v, G) &= \texttt{assume}(R(v, G)(P)) \\
\mathit{update}(x := e, v, G) &= \texttt{assume}(W(v, G)(x) = R(v, G)(e)) \\
\mathit{update}(\texttt{havoc}(x), v, G) &= \texttt{assert}(\texttt{true})
\end{aligned}
$$

$$
\begin{aligned}
\mathit{copy} &: \mathit{Vertex} \times \mathit{Vertex} \times \mathit{Graph} \to (\mathit{Vertex}, \mathit{Instruction}) \\
\mathit{copy}(u, v, G) &= (v', \{ \texttt{assume}(x_\imath = x_\jmath) \mid x \in \mathit{dom}(R) \cap \mathit{dom}(W) \wedge \\
&\qquad \imath = R(v, G)(x) \wedge \\
&\qquad \jmath = W(u, G)(x) \wedge \\
&\qquad \imath \neq \jmath \}) \text{ where } v' \text{ is fresh in } G
\end{aligned}
$$

$$
\begin{aligned}
\mathit{passivate} &: \mathit{Graph} \to \mathit{Graph} \\
\mathit{passivate}(G) &= (V', E') \\
\text{where } V' &= \{ (v, \mathit{update}(\imath, v, G)) \mid (v, \imath) \in V \} \\
&\cup \{ \mathit{copy}(u, v, G) \mid (u, v) \in E \wedge \\
&\qquad W(u, G) \neq R(v, G) \} \\
E' &= \{ (u, v) \mid (u, v) \in E \wedge \\
&\qquad W(u, G) = R(v, G) \} \\
&\cup \{ (u, v'), (v', v) \mid (u, v) \in E \wedge \\
&\qquad W(u, G) \neq R(v, G) \wedge \\
&\qquad (v', \imath) = \mathit{copy}(u, v, G) \} \\
G &= (V, E)
\end{aligned}
$$

**Figure 4.7:** Passivation of $IR_{\leftrightarrows}$ Programs

must insert some number of copy instructions to copy the incarnations of the variables written at $u$ to the incarnations read at $v$. The *copy* function accomplishes this by creating a new vertex with a set of parallel assumptions for each variale that differs between the read and write maps.

Finally, the function *passivate*$(G)$ creates a passive version of the unstructured program described by $G$. It applies the *update* function to each instruction, inserts extra copy instructions where necessary, and otherwise preserves the structure of $G$.

## 4.6 Unstructured Example

We now illustrate universal reachability applied to unstructured programs through an example that mirrors the example used to demonstrate exception handling (Sec. 3.7.5). This pattern captures one of the common defect patterns we encountered during our experimental evaluation, and consists of a section of cleanup code at the end of a function that makes certain assumptions about the state of the program before it is executed. Often, the cleanup code will release resources acquired earlier in the function. However, sometimes the function will branch to the cleanup code if acquisition of one of these resources fails.

```
void unstructured() {
    int x, y;
    x = create();
    y = create();
    acquire(x);
    if (error) goto cleanup;
    acquire(y);
    if (error) goto cleanup;
    // Use x and y
cleanup:
    release(y);
    release(x);
}
```

**Figure 4.8:** Unstructured Program Example (C)

In Fig. 4.8, we show the C source code for a small function that exhibits incorrect cleanup code, and in Fig. 4.9 we show the translation to $IR_{\leftrightarrows}$. If the first call to acquire fails, setting the global variable error, the cleanup code will immediately attempt to release y, which is still in the uninitialized state.

Before reasoning about this program, we need to specify the semantics of the create, acquire, and release functions. We do this by introducing a ghost state variable,

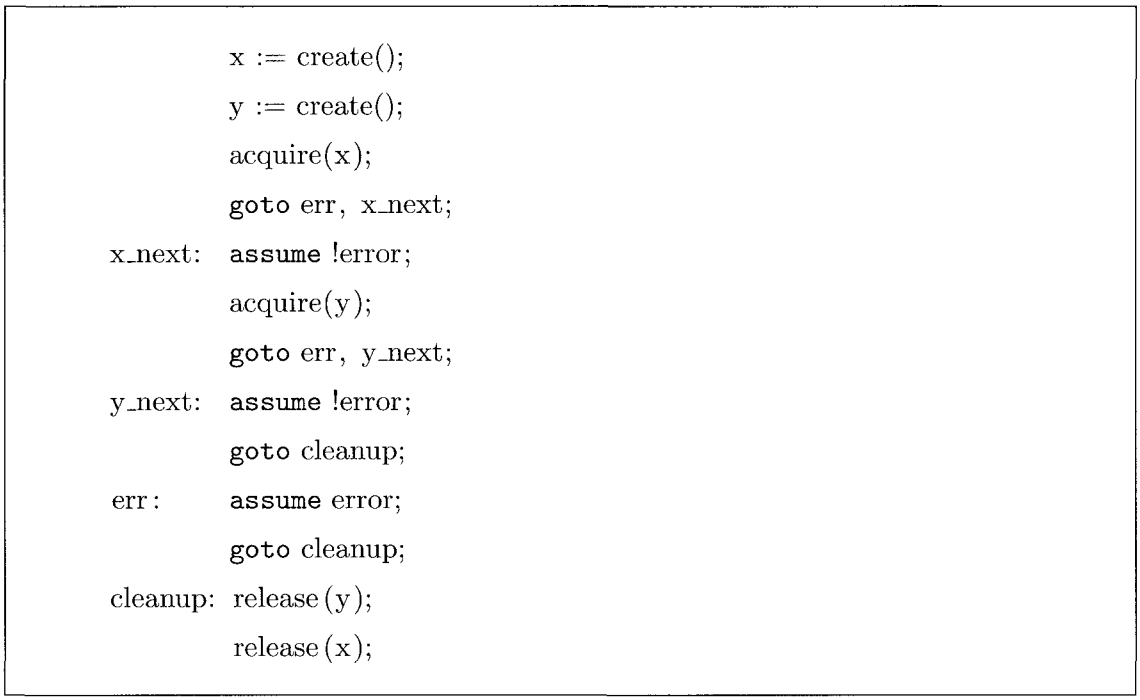


**Figure 4.9:** Unstructured Program Example ($IR_{\leftrightarrow}$)

$S_x$, for each variable $x$ that stores the result of create. We can then replace each function with a specification, in terms of assertions and assignments involving $S_x$ and $S_y$, yielding the following program.

```
        S_x := initial;
        S_y := initial;
        S_x := acquired;
        goto err, x_next;
x_next: assume !error;
        S_y := acquired;
        goto err, y_next;
y_next: assume !error;
        goto cleanup;
err:    assume error;
        goto cleanup;
cleanup: assert S_y = acquired;
```

$S_y$ := *released*;
**assert** $S_x$ = *acquired*;
$S_x$ := *released*;

We can identify slices by looking for `goto` terminators with more than one target. The first has targets "err" and "x_next". The second has targets "err" and "y_next", both of which can only be taken if he first follows the "x_next" branch. If we take the "err" branch of the first `goto`, on the other hand, the cleanup section will fail, because it begins by asserting that the ghost variable $S_y$ has the value *acquired*, but it is guaranteed to have the value *initial* at this point.

Concretely, this slice is as follows. We label each instruction numerically so that we can refer to each vertex individually within the unstructured weakest precondition derivation as $v_n$. We introduce normal and exceptional termination vertices labelled 11 and 12, respectively.

1: $S_x$ := *initial*; **goto** 2;
2: $S_y$ := *initial*; **goto** 3;
3: $S_x$ := *acquired*; **goto** 4;
4: **skip**; **goto** 5;
5: **assume** error; **goto** 6;
6: **skip**; **goto** 7;
7: **assert** $S_y$ = *acquired*; **goto** 8;
8: $S_y$ := *released*; **goto** 9;
9: **assert** $S_x$ = *acquired*; **goto** 10;
10: $S_x$ := *released*; **goto** 11;
11: **skip**
12: **skip**

We can then use the rules from Fig. 4.5 to derive a precondition $P_n$ for each vertex $v_n$,

given postconditions $N = \texttt{false}$, $X = \texttt{false}$, and $W = \texttt{true}$.

$$
\begin{aligned}
P_{12} &= \texttt{false} \\
P_{11} &= \texttt{false} \\
P_{10} &= wp(S_x := released, P_{11}, \texttt{false}, \texttt{true}) = \texttt{false} \\
P_9 &= wp(\texttt{assert}(S_x = acquired), P_{10}, \texttt{false}, \texttt{true}) \\
&= (S_x \neq acquired) \\
P_8 &= wp(S_y := released, P_9, \texttt{false}, \texttt{true}) \\
&= (S_x \neq acquired) \\
P_7 &= wp(\texttt{assert}(S_y = acquired), P_8, \texttt{false}, \texttt{true}) \\
&= (S_y = acquired) \Rightarrow (S_x \neq acquired) \\
P_6 &= wp(\texttt{skip}, P_7, \texttt{false}, \texttt{true}) = P_7 \\
P_5 &= wp(\texttt{assume}(\text{error}), P_6, \texttt{false}, \texttt{true}) \\
&= \text{error} \Rightarrow ((S_y = acquired) \Rightarrow (S_x \neq acquired)) \\
P_4 &= wp(\texttt{skip}, P_5, \texttt{false}, \texttt{true}) = P_5 \\
P_3 &= wp(S_x := acquired, P_4, \texttt{false}, \texttt{true}) \\
&= \text{error} \Rightarrow ((S_y = acquired) \Rightarrow (acquired \neq acquired)) \\
P_2 &= wp(S_y := initial, P_3, \texttt{false}, \texttt{true}) \\
&= \text{error} \Rightarrow ((initial = acquired) \Rightarrow (acquired \neq acquired)) \\
&= \texttt{true} \\
P_1 &= wp(S_x := initial, P_2, \texttt{false}, \texttt{true}) \\
&= P_2 \\
&= \texttt{true}
\end{aligned}
$$

As in the structured examples, the valid slice precondition indicates that this slice is not terminable.

# Chapter 5

# Extensions

The approach of universal reachability analysis described so far is capable of detecting a variety of bugs. However, we can improve its power with several heuristic extensions. These extensions improve universal reachability in two directions: by increasing the range of defects it can discover, and by decreasing the number of uninteresting inconsistencies it reports.

## 5.1 Boundary Conditions

The slicing function presented in Sec. 3.2 creates one slice for each alternative branch in the program. Each slice represents the subset of the program's state space that is possible if the chosen branch is taken. We can then ask whether or not it is possible for that slice to terminate successfully.

However, other subsets of the program state space can be interesting, as well. One particularly common type of error occurs when comparisons are "off by one" and therefore succeed on values that are either just below or just above the correct boundary. It turns out that our existing slicing technique applies with few modifications to this situation, as well.

Consider the program in Fig. 5.1 which clears the contents of an array (as well as one element outside of the array). One of the slices of this program will include the following fragment, corresponding to the execution of the loop body while the loop condition is true:

```
void overflow(int n) {
    int a[n];
    int i;
    for(i = 0; i ≤ n; i++) {
        a[i] = 0;
    }
}
```

**Figure 5.1:** Simple Array Clearing Program

```
assume(i ≤ n);
a[i] = 0;
```

For programming languages such as C that have only deterministic branches, we can associate a condition with every slice. Because we have one slice for every pair of source and target of a conditional branch, the condition associated with a slice is the condition that causes that branch to be taken. In the above case, the slice condition for the slice described above is i ≤n.

If we assume that errors are common on the boundary conditions of inequalities, we may want to consider the slice that represents a smaller, more targeted subset of the state space. Consider, for instance, a slice identical to the one above, but with the condition replaced with i = n. If we check the terminability of this new slice (assuming that array accesses are guarded with bounds checks), we will find that it will always fail, accessing an array element one past the end of the array.

Formally, we modify the definition of the slicing function by extending the rule for choice statements in the case where the immediate sub-statements begin with assumptions. Fig. 5.2 shows the modified case for $slices(C, s)$ function, along with a pair of auxiliary functions.

As we will see in Sec. 7.8, this approach is particularly good at detecting buffer overflows. However, it also has some disadvantages, and results in false positives in a number of common circumstances. These are "real" false positives in a sense, as they don't represent violations of universal reachability for the overall statement. Consider,

$$slices(C, s_1 \ \square \ s_2) = slices(C, s_1) \cup slices(C, s_2) \cup bslice(C, s_1) \cup bslice(C, s_2)$$

$$bslice \quad : \quad Context \times Statement \rightarrow 2^{Statement}$$

$$bslice(C, s) = \begin{cases} \{C[\texttt{assume}(bcond(P)) \ ; \ s']\} & \text{if } s = \texttt{assume}(P) \ ; \ s' \\ & \text{and } bcond(P) \text{ is defined} \\ \emptyset & \text{otherwise} \end{cases}$$

$$bcond \quad : \quad Predicate \rightarrow Predicate$$

$$bcond(e_1 < e_2) = (e_1 = e_2 - 1)$$

$$bcond(e_1 > e_2) = (e_1 = e_2 + 1)$$

$$bcond(e_1 \leq e_2) = (e_1 = e_2)$$

$$bcond(e_1 \geq e_2) = (e_1 = e_2)$$

**Figure 5.2:** Boundary Condition Slicing

for instance, the code in Fig. 5.3 that performs different operations for zero, positive numbers, and negative numbers.

```
if(x == 0) {
  ...
} else if (x > 0) {
  ...
} else {
  ...
}
```

**Figure 5.3:** Example of Boundary Condition Dead Code

This program fragment has two choice statements, one with a branch for x == 0 and a branch for x $\neq$0, and one with a branch for x > 0 and a branch for x $\leq$0. Note, however, that the latter two branches operate in a context where the x $\neq$0 branch has been taken, so x $\neq$0 is necessarily true. Therefore, when we take the x $\leq$0 slice and specialize it to x == 0, we get a contradiction that does not represent a real

error.

The next section describes a technique that can be used to avoid some warnings that arise either from dead code, or that arise as a result of this problematic case for boundary condition slicing.

## 5.2 Buffer Overflow Example

Here we show a complete derivation of the example from the previous section in which boundary condition slicing can lead to the detection of a buffer overflow. Consider the example from Sec. 5.1, reproduced in Fig. 5.4. It consists of a simple loop that fills an array with zeros, and then proceeds to write to an element past the end of the array.

```
void overflow(int n) {
    int a[n];
    int i;
    for(i = 0; i ≤ n; i++) {
        a[i] = 0;
    }
}
```

---

```
i := 0;
(   assume i ≤ n;
    a[i] := 0;
    i := i + 1
  □
    assume i > n )*
```

**Figure 5.4:** Buffer Overflow Example

Here, the interesting slice is the one in which the loop body executes. Following the slicing function from Sec. 3.2, we slice the loop body and sandwich it between two

copies of the entire loop.

$$
\begin{array}{l}
i := 0; \\
((\texttt{assume}(i \leq n); a[i] := 0; i := i + 1) \ \square \ \texttt{assume}(i > n))^*; \\
(\texttt{assume}(i \leq n); a[i] := 0; i := i + 1); \\
((\texttt{assume}(i \leq n); a[i] := 0; i := i + 1) \ \square \ \texttt{assume}(i > n))^*
\end{array}
$$

This slice has the slice condition $i \leq n$, which we specialize according to the rules in Sec. 5.1 to $i = n$. The modified slice is therefore as follows.

$$
\begin{array}{l}
i := 0; \\
((\texttt{assume}(i \leq n); a[i] := 0; i := i + 1) \ \square \ \texttt{assume}(i > n))^*; \\
(\texttt{assume}(i = n); a[i] := 0; i := i + 1); \\
((\texttt{assume}(i \leq n); a[i] := 0; i := i + 1) \ \square \ \texttt{assume}(i > n))^*
\end{array}
$$

We can use a very coarse over-approximation of loops, applying havoc to all of the targets of the loop body, and still detect an error.

$$
\begin{array}{lll}
 & wp( & i := 0; \\
 & & \texttt{havoc}(i, a); \\
 & & \texttt{assume}(i = n); \\
 & & \texttt{assert}(i < n); \\
 & & a[i] := 0; \\
 & & i := i + 1; \\
 & & \texttt{havoc}(i, a), \qquad \texttt{false}, \texttt{false}, \texttt{true}) \\
= & wp( & i := 0; \\
 & & \texttt{havoc}(i, a); \\
 & & \texttt{assume}(i = n); \\
 & & \texttt{assert}(i < n), \quad \texttt{false}, \texttt{false}, \texttt{true}) \\
= & wp( & i := 0; \\
 & & \texttt{havoc}(i, a); \\
 & & \texttt{assume}(i = n), \quad i \geq n, \texttt{false}, \texttt{true}) \\
= & wp( & i := 0; \\
 & & \texttt{havoc}(i, a), \qquad i = n \Rightarrow i \geq n, \texttt{false}, \texttt{true}) \\
= & i' = n \Rightarrow i' \geq n & \\
= & \texttt{true} &
\end{array}
$$

Thus the original loop test i $\leq$ n evaluates to `true` when i == n, but in this case the loop body will not be terminable. This divergence strongly suggests that loop body test i $\leq$ n is problematic, and indeed it results in writing past the end of the array a.

Showing the out-of-bounds array index requires knowing the bounds of the array a. In this case, it is local to the procedure under analysis, so the bounds are known. On larger programs, inter-procedural propagation of possible array bounds would be necessary. This could happen via a separate analysis pass, or via precise procedure specifications.

## 5.3 Avoiding Dead Code Warnings

Previously, we have stated that violations of universal reachability indicate either assertion failures or dead code. Under some circumstances both are interesting and indicate problems in the underlying code. In other situations, however, dead code is inevitable or at least relatively uninteresting.

Fortunately, it is sometimes possible to distinguish assertion failures from unreachable code when using weakest preconditions to check terminability, in one of two ways.

First, we have so far always used the *wp* function with the $W$ postcondition equal to `true`. If $wp(s, \texttt{false}, \texttt{false}, \texttt{true})$ is valid, the statement will either fail an assertion or assumption. However, we can also calculate the weakest precondition with the $W$ postcondition equal to `false`. If $wp(s, \texttt{false}, \texttt{false}, \texttt{false})$ is valid, then the statement contains dead code.

Alternatively, assume we have a function $wp'$ that behaves exactly like *wp* but ignores assertions (*i.e.*, treats them all as `assert(true)`). If $wp'(s, \texttt{false}, \texttt{false})$ is valid, then $s$ must be non-terminable due to assumptions alone.

These two approaches correspond to two different interpretations of the term "dead code", and which to use may be a matter of preference. We can show that $wp(s, \texttt{false}, \texttt{false}, \texttt{false}) \Rightarrow wp'(s, \texttt{false}, \texttt{false})$, but the opposite implication does not always hold.

Therefore, we can first check terminability using a $W$ postcondition of `true`, and then, if the result is valid, use either of the weakest precondition calculations de-

scribed in the previous paragraphs. Only if the first is valid and the second is not do we report an inconsistency.

Because the false positives that sometimes arise from boundary condition slicing are always the result of inconsistencies between assumptions, and do not involve assertions, this technique can help in that case, as well. For instance, it will eliminate the spurious warning for the example given in Fig. 5.3.

## 5.4 Avoiding Duplicate Warnings

One final heuristic helps minimize the number of warnings generated by universal reachability analysis, without leading to false negatives. The key is that, if an entire function body is non-terminable, the same will be true of all of its slices. In addition, if a slice is non-terminable, the same will be true of any slices refined to check boundary conditions. Therefore, an implementation can first check terminability of the entire body of a function. If the body is terminable, then the analyzer can calculate each of the function's slices, and check terminability of each. If a given slice is terminable, the analyzer can then check a boundary condition variant of that slice, if one exists.

Not only does this approach reduce the number of redundant warnings generated by the analysis, it also makes the entire process more efficient, because slicing, failure condition generation, and satisfiability checking occur only if their results will be useful.

# Chapter 6

# Implementation

To validate the utility of universal reachability analysis, we developed a prototype implementation called Curate that can check for inconsistencies in C programs. Curate is implemented in a little under 5000 lines of Haskell, using the `language-c` package [54] to parse and type-check C, a custom translator and weakest precondition generator for guarded commands, and the Yices SMT solver [60] to answer validity queries.

This chapter describes the design decisions, features, and limitations of Curate. The following chapter presents the results of evaluating the implementation on a large collection of open source software.

## 6.1 Language Choice

We chose to focus on the C language because it is widely used and subject to a number of common errors due to undefined behaviors in its specification. The presence of undefined behavior in the language definition makes software development in C more error-prone, but it also makes universal reachability analysis useful in a push-button manner on unannotated programs because we can automatically insert assertions that require the programs to fall within the well-defined subset of the language.

The use of C also makes it easier to compare our results to those of other defect-detection tools. The majority of open source defect-detection tools also target C, and almost every currently available commercial tool can analyze C as well.

## 6.2 Architecture

Curate begins by transforming C into an intermediate form essentially the same as the $IR_{\rightsquigarrow}$ language presented in Sec. 4.1, using a transformation similar to that presented in Sec. 4.2. The essential differences occur in function call instructions. In Curate, the function call instruction does not introduce a branch to an exceptional termination vertex (because C lacks exceptions), and function calls can take parameters and assign a result value.

Once the original C source has been translated into the intermediate language, we eliminate loops and insert the appropriate havoc instructions, as described in Sec. 4.3.1. We then translate the loop-free function bodies into passive form, using the algorithm from Sec. 4.5.1. Next, we slice each function body according to the algorithm in Sec. 4.4, yielding a set of slices $s_1 \ldots s_n$. We then calculate the weakest precondition of each slice according to the algorithm given in Sec. 4.3. Finally, we use Yices to check the satisfiability of the negation of this precondition, and report an inconsistency if Yices reports that the negated precondition is unsatisfiable.

## 6.3 Memory Model

We treat the entire heap of the program as a single array, mapping unbounded integers to unbounded integers. Local variables become variables in the syntax of the intermediate representation, unless their address is taken, in which case we introduce a variable representing the address of the original variable, and use the address variable to index into the heap array.

This memory model is simple to implement, but has drawbacks. Values that overlap in memory are not treated correctly. For example, if an array of characters $a$ and an integer $i$ are aliased, our memory model will treat $a[0]$ and $i$ as having equal values, while the remaining elements of $a$ will have no connection to the value of $i$.

We have not noticed false positives caused by this approximation in our experimental results, but we expect it may lead to false negatives.

## 6.4 Loops

Loops introduce back edges in the control flow graph. As described earlier, we remove back edges and insert a havoc instruction at each loop head. This havoc instruction overwrites any variable written in the loop body. We calculate the loop body by determining the set of instructions that are reachable from the target of the back edge and that can reach the source of the back edge, as described in Sec. 4.3.1. This obscures which iteration is currently under analysis. Assertions can be used to describe any loop invariant that can be written as a C expression.

When we calculate the assignment targets of the body of a loop, we consider the heap array as a single variable. Therefore, any writes to the heap within a loop body will cause the entire value of the heap to be lost. With extra engineering effort, it would be possible to limit the range of these havoc instructions to the addresses in the heap actually affected by the loop body, retaining the value of the remaining portions of the heap.

## 6.5 Procedures

We perform universal reachability analysis within the body of a single procedure at a time. An earlier, abandoned implementation made some steps toward inter-procedural analysis. The earlier implementation was complex an inefficient, so we focused later development efforts on intra-procedural analysis. Future work exploring the best approach to inter-procedural analysis might be valuable.

To prevent false positives related to procedure side-effects, we insert a havoc instruction after each function call that replaces the entire heap with a fresh array. With significant extra engineering effort, it would be possible to perform an effect analysis and havoc only those portions of the heap that the function may actually modify.

## 6.6 Decision Procedure

Curate uses the Yices SMT solver [60], version 1.0.28, to decide the validity of the failure conditions generated by weakest precondition analysis. Because Yices

checks input formulas for satisfiability, rather than validity, we simply negate the failure conditions before checking them.

## 6.7 Limitations of C Support

Curate translates C source code into an internal representation that essentially matches the $IR_{\leftrightarrow}$ language presented in Ch. 4. During this translation we attempt to capture the entire semantics of the input C program. However, there are a few cases where the translation either over- or under-approximates the semantics of the original program. There are known techniques for avoiding all of the following limitations but, in the context of establishing a proof-of-concept, the benefits of the more complex approaches were outweighed by the sometimes significant added costs of engineering effort or computational expense.

**Finite-precision integers.** We use the unbounded `int` data type in the Yices SMT solver to model all integral C data types. Therefore, we will not correctly model integer overflow or underflow. This could lead to either false positives or false negatives. The use of either modular arithmetic or a bit-vector model could alleviate this problem. However, Yices has no support for modular arithmetic, and bit-vector arithmetic is significantly less efficient.

**Memory model.** Curate interprets the memory of the C program as a single map from unbounded integer address to unbounded integer values. For most cases, this treatment works well. However, if two objects overlap in such a way that they refer to the same memory address with types of different sizes, this interpretation can lead to either false positives or false negatives. An example of the problematic nature of this approach is a union of an integer with a structure of four characters. In a correct implementation, the first character would be equivalent to either the least significant or the most significant byte of the integer, depending on the target architecture. In our interpretation, the first character will have a value identical to the value of the integer, and the values of the other three characters will be unrelated to the value of the integer. Many more precise memory models exist [62, 23], and a production-quality implementation would benefit from using

one of them. However, for our prototype, the added complexity outweighed the benefit.

**Floating point.** Curate treats floating-point constants as unconstrained and floating-point operations as uninterpreted functions. This could lead to false negatives, but does not lead to false positives. In our case, the decision to ignore floating-point operations is largely due to the lack of support for floating-point arithmetic in Yices, our chosen decision procedure. However, decision procedures with support for floating-point arithmetic do exist [67].

**Non-linear arithmetic.** Curate treats non-linear arithmetic operations as uninterpreted functions. This is largely due to the lack of support in Yices for reasoning about non-linear arithmetic.

**Bitwise operators.** Curate treats all C operators that work on individual bits (|, &, ^) as uninterpreted functions. This could lead to false negatives, but should not lead to false positives. We could avoid this limitation by using a bit-vector model, either globally or selectively in cases where bit-vector operations occur. For simplicity and efficiency, we have not taken this approach in our prototype, however.

**Multi-dimensional arrays.** Curate does not support multi-dimensional arrays, as they occur rarely and are more complex to model than one-dimensional arrays. Fortunately, we encounter these very rarely in our benchmarks. This could lead to either false positives or false negatives.

**Indirect jumps.** We treat any `goto` statement with a pointer target as if it were a return statement. This could lead to false negatives, but should not lead to false positives. We could avoid this limitation with a value analysis of some sort. It might require whole-program abstract interpretation in the worst case. Indirect jumps are rare, so the additional effort to support them would not be justified for a prototype tool.

**Inline assembly.** We make use of the annotations on inline assembly fragments that indicate which C variables may be modified, but do not analyze the assembly in-

structions themselves. This could lead to false negatives, but should not lead to false positives. This limitation is not intrinsic in any way. Supporting inline assembly would simply require us to be able to translate arbitrary assembly instructions into our intermediate language, but this would cost significant engineering effort.

# Chapter 7

# Experimental Results

We will now describe the results of running our implementation of universal reachability analysis, Curate, along with two other defect-detection tools, on a collection of benchmarks of varying sizes. We measured the number and type of defects found by each tool, how these defects overlap between tools, and the rates and causes of false positives. We also measured the time taken for each tool to complete its analysis, along with some Curate-specific measurements on the effects of slicing and passivation. Our ultimate goal is to determine how practical and useful universal reachability analysis is in comparison with existing tools.

We compared Curate with the Saturn null pointer analysis and the Clang Analyzer. The results show that Curate runs in less time than the comparable Saturn null pointer analysis, while usually finding a larger number of inconsistencies, although Saturn does discover one class of inconsistency that Curate does not. The Clang Analyzer runs much more quickly than either Curate or Saturn, but discovers primarily shallow bugs, most of which will not lead to run-time errors. The warnings given by Clang only overlap with either Saturn or Curate in a handful of cases.

The following sections provide a detailed discussion of these findings, and conclude with a summary of where each tool could most usefully be applied.

## 7.1 Benchmarks

We chose a collection of several programs, large and small, on which to evaluate how well universal reachability analysis can detect bugs. Because of the similarity between our analysis and that presented in Dillig, Dillig, and Aiken's PLDI 2007 paper [37], we included a subset of the benchmarks from their paper in our collection. These benchmarks consist of:

- MPlayer 1.0pre8, a media player for a wide variety of audio and video formats.
- OpenSSL 0.9.8b, a library implementing the Secure Socket Layer (SSL) protocol, along with the necessary cryptographic algorithms.
- Samba 3.0.23b, a set of programs for interacting with shared file systems that use the Server Message Block (SMB) protocol.
- OpenSSH 4.3p2, an implementation of the Secure SHell (SSH) protocol.
- Sendmail 8.13.8, a widely-used mail transport agent (MTA).

We skipped the Pine and Linux benchmarks included in the original paper by Dillig *et al.* because they presented parsing or type-checking difficulties to one or more of the tools included in our comparison. For the Samba benchmark, we give only the total number of warnings produced by each tool, as the combined number of warnings from the three tools was prohibitively high for manual categorization.

These programs range from around 900,000 lines of code (LOC) to over 3 million LOC in size after pre-processing. While Dillig *et al.* discovered a number of inconsistencies in each, these are all well-tested and widely-used programs whose defect density is likely to be relatively low.

Next we include a set of widely-used programs chosen independently. Some of these are known to have particular bugs. These benchmarks include:

- BC 1.06, a command-line calculator program.
- SpiderMonkey 1.70, the JavaScript interpreter used in the Firefox web browser (written as `js` for short).

- NCompress 4.2.4, a file compression utility.
- PCC 0.9.9, a simple C compiler.
- Squid 2.3STABLE1, a caching proxy server for the HTTP protocol.
- LibTIFF 3.7.0, a library for reading and writing Tagged Image File Format (TIFF) files.

Finally, we include a set of small programs, almost all of which contain known bugs, published by the SAMATE project [79] of the National Institute of Standards and Technology (NIST) with the purpose of benchmarking automated defect-detection tools. This collection consists of more than 1500 simple C files that contain isolated instances of common security bugs. Some are hand-written, and some are extracted from open-source programs.

## 7.2 Tools

The goal of our experimental evaluation is to determine how universal reachability analysis compares to existing defect-detection mechanisms. Because most of our benchmark programs are fairly large and make use of a wide variety of C language features and extensions, we limit ourselves to analysis tools that can parse at least most of the source code included in our benchmarks, and that can complete their analysis on the majority of the benchmarks without running out of memory or exceeding pre-determined time bounds.

Based on these considerations, we have chosen two tools to compare against Curate:

**The Saturn null Pointer Analysis** This is the analysis described by Dillig *et al.* that shares many characteristics with universal reachability analysis [37]. Our experiments use the version included in Saturn 1.1. Like our work, this tool attempts to discover program inconsistencies. Unlike our work, their implementation focuses specifically on detecting null pointer dereferences, and therefore reasons only about pointer values. A more detailed comparison with our analysis appears in Sec. 8.1.

**The Clang Analyzer** This tool was developed as part of the Low-Level Virtual Machine (LLVM) project, and performs a number of simple and efficient static analysis techniques [83]. Our experiments are based on build 247. It is intended to be a practical tool, usable within a wide range of real development environments, and has been integrated into Apple's XCode IDE. Because this tool emphasizes practicality, we place special emphasis on comparing its warning rate and performance with that of our analysis.

## 7.3 Experimental Setup

Before running the benchmark programs through each of the analysis tools, we set up an experimental framework to ensure that we could reasonably compare the results between benchmarks and between tools.

### 7.3.1 System Configuration

We performed all experimental evaluation on a workstation running Mac OS X 10.6.5 with 8GB of RAM and a 4-core 2.5GHz Intel i5 processor.

### 7.3.2 Pre-processing

C programs typically make use of the C pre-processor (CPP) to allow for inclusion of external files, macro definitions, and conditional compilation. One of the primary purposes for these features is to allow C code to be portable across a wide range of programs. A consequence of this, however, is that the actual C program that reaches a compiler or static analysis tool is highly dependent on the environment in which it is pre-processed.

Two of the three defect-detection tools listed in the previous section can automatically pre-process source code before analysis, but this raises the possibility that each tool may specify slightly different macro definitions during pre-processing, resulting in a slightly different program. To mitigate this problem, we perform an initial phase where we pre-process each program once, and use the output of this single pre-processor run as the input to each of the analysis programs.

### 7.3.3 Analysis Parameters

**Exit Functions** One critical consideration for both Saturn's null pointer analysis and universal reachability analysis is the detection of exit functions. The implementation of the `assert` function in C often consists of a conditional that aborts the program if the assertion evaluates to `false`, and continues execution otherwise. Knowing that the function call that terminates the program never returns is essential to avoiding false positives for both analyses.

Saturn includes an analysis to detect exit functions, and the result of this analysis, if available, is automatically used by the null pointer analysis. Therefore, we ran Saturn's exit function analysis as a pre-processing phase on all of the benchmarks, and used its results as input to Curate, as well.

In several cases, exit functions not detected by Saturn's exit function analysis were relevant to the results produced by either Saturn or Curate. We manually filtered out warnings due to undetected exit functions in either of these tools. The Clang Analyzer has no way to indicate exit functions, so we used its results without filtering.

**Time Limits** We ran both Saturn and Curate with a timeout value of 120 seconds. For Saturn, no function body analysis is allowed to take longer than this limit. For our tool, no individual slice analysis can take longer than this limit (including the original slice representing the entire function body). The Clang Analyzer easily completes analysis of every function, and indeed every file, in less than this time limit, and does not have an option for setting a time limit.

### 7.3.4 Source Code Changes

Because each tool uses a different front end to parse and type-check the C source code, we needed to make some minor modifications to the source code, after pre-processing, so that all tools could successfully load it. The modifications we made fell into the following categories:

- Adding function prototypes.
- Commenting out some complex static global initializers (which should be irrelevant

to all three tools).

- Removing prototype parameters with the name __block, since that has become a keyword in Clang.
- Commenting out a few small sections of very non-portable C code in MPlayer so all three front ends could process it.
- Limiting the set of files processed to include only one main function, since Saturn does a whole-program analysis. Many of the benchmarks we chose consisted of a common library with a set of programs that make use of it, and we had to choose a single one of these programs to take part in the analysis.
- Making other small changes to get parsing and type checking to succeed for all three front ends.

## 7.4 Warning Categories

We classified each warning produced by any of the three tools into one of the following categories. The first collection of categories represents likely bugs, or at least awkward code. The second collection represents various types of dead code that can be detected by universal reachability analysis. The third represents false positives, largely caused by engineering tradeoffs in each of the tools. For each category, we give an abbreviation used in the tables of experimental results. For a few cases, we were not able to determine the cause of the inconsistency warning (typically due to the complexity of the associated source code). We place these few warnings in the "Unknown" category.

### 7.4.1 True Errors

We refer to warnings that indicate bugs or awkward programming practices as *true* errors.

**Bug-Causing Inconsistency (Inc. Error)** The most interesting inconsistency warnings indicate cases where some path through the function under analysis could lead to either an assertion failure or an incorrect result. This category includes any

indication that an assertion failure is certain within a normal slice (as opposed to a boundary condition slice, as described next), even if the rest of the program never calls the function under analysis with arguments that would lead to a crash. It also includes cases where an assertion failure may not occur, but where the code will clearly operate incorrectly. When the Clang Analyzer discovers a potential null pointer dereference that coincides with one discovered by either Saturn or Curate, we include it in this category.

**Boundary Condition Inconsistency (Boundary)** If Curate discovers a potential assertion failure using boundary condition specialization, as described in Sec. 5.1, it falls into this category. Neither of the other tools ever produces warnings in this category.

**Programmer Confusion (Confusion)** Some cases of dead code detected by Curate or Saturn will not lead to assertion failures, but clearly indicate confusion or misunderstanding on the part of the author of the function under analysis. Warnings in this category indicate code that would benefit from improvement. This category includes cases where dead code exists because a feature has not been completely implemented, and therefore parts of the implementation are temporarily unreachable. We believe it is desirable to point out these cases, to help remind programmers of remaining work that they may have forgotten. In short, this category consists of any dead code that we consider to be a defect, but that will not directly cause incorrect behavior or an assertion failure.

**Other Tool Warning (Other)** Saturn's null pointer analysis includes a number of heuristics that fall outside of the scope of universal reachability analysis. In addition, it performs an inter-procedural analysis, while Curate is intra-procedural. The Clang Analyzer also produces warnings for a number of problems, such as values written but never read, that are outside of the scope of our comparison. The Clang warnings are generally correct, but superficial. We group all of these warnings together. When the Saturn null analysis or Clang generate warnings that coincide with those generated by universal reachability analysis, we classify them as inconsistencies.

### 7.4.2 Dead Code

Some cases of inconsistency indicate dead code that is intentional or unavoidable, such as that which arises in code intended to run on multiple platforms. We separate these from the true errors just described.

**Trivial Impossible Cases (Trivial)** Our technique for translating loops results in one branch that assumes the loop condition and one branch that assumes the negation of the loop condition. For infinite loops, the loop condition is often simply `true`, leading to a branch that assumes `false`. This leads to a trivial inconsistency, but one that is easy to detect using the techniques of Sec. 5.3. In addition, some conditionals include an implicit, impossible "else" branch that does not represent an error. For example, in the code `if` (c) { ... } `else if` (!c) { ... }, either c or !c must be true, so the implicit else branch of the second conditional will never be executed.

**Configuration Dead Code (Configuration)** Some dead code occurs as a result of comparisons between values that are constant at compile time but that may vary across different system configurations. For example, it is common to compare the size of a particular type with various constants to determine the capabilities of the current system architecture. In one interpretation, these are true inconsistencies, but it would be desirable to avoid warnings about them. Conditional compilation through the inclusion or omission of code by the pre-processor can also lead to dead code.

**Macro Instantiation (Macro)** Macros, like functions, typically represent abstractions and are written to apply in a wide variety of situations. To accomplish this, their expansions often contain conditional statements. In any particular context, some slices of the expansion of a macro may contain dead code, and therefore lead to inconsistency warnings. These warnings could be avoided in an implementation that integrated the pre-processor into the parser and could therefore track macro expansions and treat them like function calls when possible. As a work-around, Curate automatically filters out warnings regarding branches that begin and end on the same line. This approach has the unfortunate side-effect of obscuring some true errors, and does not entirely eliminate

spurious warnings caused by macros. It does, however, significantly reduce the number of macro-related warnings.

### 7.4.3 False Positives

Finally, some warnings reported by our tool are false positives in that they do not represent violations of universal reachability when considering the full semantics of the language. These occur due to heuristics, engineering compromises, or potentially bugs in the implementation.

**Boundary Condition False Positive (Boundary FP)** As described in Sec. 5.1, the boundary condition extension to our analysis can result in false positives when a specialized slice condition implies a condition already covered by an earlier branch.

**Unsupported (Unsupported)** Some warnings occur as the result of explicitly unsupported language features. In our case, this includes multi-dimensional arrays, floating point arithmetic, non-linear arithmetic, bit-level operations, overflow, and conversion between signed and unsigned values. These warnings would be avoidable with a more complete implementation. We also include in this category some cases where programmers intentionally stretch the boundaries of the semantics of C, but in ways that cannot cause runtime failures. Examples include taking the address of a field of a dereferenced null pointer (*e.g.*, &(((`struct` s *)NULL)→fld)) to calculate the offset of a field, and taking the address of an array element at an out-of-bounds index (*e.g.*, &(a[`sizeof`(a)])) but never dereferencing it.

## 7.5 Small Examples

We first ran all three tools on the collection of small examples presented in Sec. 3.7, Sec. 5.2 and Sec. 4.6. Each of these has only one or two known bugs, and each is intended primarily as an illustration of a program inconsistency. In Tbl. 7.1 we show the number and category of warnings produced by each of the three tools.

As these examples were written to illustrate errors detectable using universal reachability analysis, Curate generates warnings about all of them (along with one

case of trivial dead code in the binary search example). Saturn also warned about the inconsistency in the doubly-linked list example, but the Clang Analyzer was unable to detect any of the errors.

The results on these benchmarks confirm that Curate can detect defects of the type we expect it to find, and that other tools miss. The `list` example includes an inconsistency involving a null pointer dereference, so Saturn detects it.

| Benchmark | Category | Curate | Saturn | Clang |
|---|---|---|---|---|
| `bsearch` | Inc. Error | 1 | 0 | 0 |
| | Trivial | 1 | 0 | 0 |
| `dblfree` | Inc. Error | 1 | 0 | 0 |
| `list` | Inc. Error | 1 | 1 | 0 |
| `oom` | Inc. Error | 1 | 0 | 0 |
| `overflow` | Boundary | 1 | 0 | 0 |
| `unstructured` | Inc. Error | 1 | 0 | 0 |
| `useafterfree` | Inc. Error | 1 | 0 | 0 |
| Total | All | 8 | 1 | 0 |

Table 7.1: Warnings from Small Examples

## 7.6 Saturn Benchmarks

Next, we ran each of the tools on a subset of the benchmarks used for Saturn's null pointer analysis. The numbers of warnings each tool produces in each category appear in Tbl. 7.2. In this table, we break down the warnings with a separate row for each category under each benchmark, omitting rows in which all entries are zero. For each benchmark, we also list the number of pre-processed lines of code (PPLOC) it contains. Saturn and Clang support only a subset of the warning categories we have defined. We write a hyphen in entries that are not supported by the tool in question.

Some of the entries for Saturn and Curate are of the form $S + U$, where $S$ is the number of warnings shared between the two tools, and $U$ is the number of unique warnings generated by the tool. The entries for Clang in the "Inc. Error" category are of the form $S + C + B$, where $S$ is the number shared with Saturn, $C$ is the number shared with Curate, and $B$ is the number shared with both.

| Benchmark | Category | Curate | Saturn | Clang |
|---|---|---|---|---|
| `MPlayer-1.0pre8` | Inc. Error | 6+27 | 6+13 | 2+3+0 |
| PPLOC: 2,552,948 | Boundary | 2 | - | - |
| | Confusion | 1+39 | 1+8 | - |
| | Other | 0 | 58 | 1149 |
| Dead code | Trivial | 4 | 0 | - |
| | Configuration | 1 | 0 | - |
| | Macro | 7 | 2 | - |
| False positives | Boundary FP | 62 | - | - |
| | Unsupported | 14 | 1 | - |
| Unknown | Unknown | 3 | 0 | - |
| `openssh-4.3p2` | Inc. Error | 1 | 0 | 0 |
| PPLOC: 907,011 | Confusion | 1 | 0 | - |
| | Other | 0 | 4 | 72 |
| Dead code | Trivial | 1 | 0 | - |
| | Configuration | 1 | 0 | - |
| | Macro | 0 | 1 | - |
| False positives | Boundary FP | 5 | - | - |
| `openssl-0.9.8b` | Inc. Error | 4+5 | 4+2 | 2+3+2 |
| PPLOC: 3,084,124 | Confusion | 36 | 6 | - |
| | Other | 0 | 138 | 517 |
| Dead code | Trivial | 6 | 0 | - |
| | Configuration | 2 | 0 | - |
| | Macro | 159 | 0 | - |
| False positives | Boundary FP | 25 | - | - |
| | Unsupported | 3 | 0 | - |
| Unknown | Unknown | 4 | 1 | 0 |
| `samba-3.0.23b` | Unknown | 834 | 219 | 320 |
| PPLOC: 10,765,507 | | | | |
| `sendmail-8.13.8` | Inc. Error | 0 | 0 | 0 |
| PPLOC: 498,941 | Confusion | 4 | 0 | - |
| | Other | 0 | 23 | 151 |
| Dead code | Trivial | 1 | 0 | - |
| | Configuration | 2 | 0 | - |
| | Macro | 3 | 0 | - |
| False positives | Boundary FP | 4 | - | - |
| | Unsupported | 8 | 1 | - |
| Unknown | Unknown | 8 | 0 | - |

Table 7.2: Warnings from Saturn Benchmarks

On this set of benchmarks, Curate detects many of the same inconsistencies as Saturn, as expected, along with several additional inconsistencies. The additional inconsistencies all fall outside of the domain of Saturn's analysis, either because they involve non-pointer variables, or because they rely on function preconditions or logical theories not included in Saturn's null pointer analysis.

Saturn also detects a few errors that Curate misses. Some of these are because Saturn's null pointer analysis includes a number of heuristics that fall outside of the domain of pure inconsistencies. These errors are listed in the "Other" category. However, Saturn also detects a few true inconsistencies that Curate misses: 13 for MPlayer and 2 for OpenSSL.

The inconsistencies detected by Saturn and not Curate occur for two reasons. The first is that the translation Curate uses for the C short-circuit operators (&&, ||), is semantically correct, but obscures inconsistencies from universal reachability analysis. We could have modified the translation of short-circuit operators to match that of the `if` statement, but we decided to maintain the existing approach because it illustrates an important disadvantage of universal reachability analysis: semantically-correct transformations that change the branching structure of a program can affect what inconsistencies are detected.

The second reason is that Saturn's analysis detects a class of inconsistencies that do not represent violations of universal reachability but still often indicate real defects. We describe this class of inconsistencies further in Sec. 7.10.

Along with a slightly higher number of true defects, Curate also reports a higher number of arguably spurious warnings. Some are false positives resulting from engineering compromises, while others simply indicate dead code. The largest collections of spurious warnings result from the use of macros, and from boundary condition specialization. Boundary condition specialization discovered 2 real errors in this set of benchmarks and 96 false positives. Excluding macros and boundary condition false positives, however, the rate of spurious or uninteresting warnings is low, and could be made lower in a production-quality implementation.

Compared to the Clang Analyzer, both implementations of inconsistency detection generate fewer warnings, except in the case of Samba. Although the effort to

investigate all inconsistency warnings can become significant for large code bases, the popularity of the Clang Analyzer suggests that the warning rate is low enough to be palatable to many programmers. Integration with an IDE for use during development, perhaps applied only to code already being modified, may be a practical approach.

## 7.7 Other Open Source Package Benchmarks

We also ran each of the tools on our additional selection of open source benchmarks. The results of this appear in Tbl. 7.3. Again, we break down the warnings with a separate row for each category under each benchmark, omitting rows in which all entries are zero.

In this collection of benchmark programs, we observe similar patterns to those in the previous set. The Clang Analyzer generates more warnings than either of the inconsistency detectors, except in the case of SpiderMonkey (`js`). Curate detects two significant inconsistencies missed by Saturn's null analysis (on SpiderMonkey), and Saturn detects two significant inconsistencies missed by Curate (on BC and PCC). Excluding macros, the number of spurious warnings is relatively low for all benchmarks.

The inconsistencies discovered by Saturn but not Curate fit into the categories described in the previous section. Each is either an artifact of the translation of short-circuit operations or falls into the category of inconsistency that falls outside of universal reachability analysis, described below in Sec. 7.10.

## 7.8 SAMATE Benchmarks

Finally, we ran each of the tools on the collection of small benchmarks from NIST's SAMATE team. The results appear in Tbl. 7.4. We break down the warnings with a separate row for each category under each benchmark, omitting rows in which all entries are zero.

In this collection, we see the benefits of boundary condition specialization. The majority of the SAMATE test cases include buffer overflows of varying complexity. Almost all of them are detectable through intra-procedural analysis, and involve local arrays of known size. This is the condition in which boundary condition specialization

| Benchmark | Category | Curate | Saturn | Clang |
|---|---|---|---|---|
| `bc-1.06` | Other | 0 | 1 | 24 |
| PPLOC: 42,607 | Boundary FP | 6 | - | - |
| False positives | Unsupported | 2 | 0 | - |
| `js-1.70` | Inc. Error | 2 | 0 | 0 |
| PPLOC: 309,428 | Boundary | 1 | - | - |
| | Confusion | 16 | 1 | - |
| | Other | 0 | 96 | 99 |
| Dead code | Configuration | 2 | 0 | - |
| | Macro | 11+28 | 11+5 | - |
| False positives | Boundary FP | 4 | - | - |
| | Unsupported | 5 | 1 | - |
| Unknown | Unknown | 0 | 3 | - |
| `ncompress-4.2.4` PPLOC: 2,915 | | No warnings | | |
| `pcc-0.9.9` | Inc. Error | 0 | 1 | 0+1+0 |
| PPLOC: 72,948 | Other | 0 | 22 | 35 |
| Dead code | Trivial | 1 | 0 | - |
| | Configuration | 2 | 0 | - |
| | Macro | 1 | 0 | - |
| False positives | Boundary FP | 4 | - | - |
| | Unsupported | 1 | 2 | - |
| `squid-2.3.STABLE1` | Inc. Error | 1+0 | 1+0 | 1+1+1 |
| PPLOC: 807,286 | Confusion | 4 | 0 | - |
| | Other | 0 | 20 | 243 |
| Dead code | Trivial | 2 | 0 | - |
| | Macro | 1+21 | +1 | - |
| False positives | Boundary FP | 5 | - | - |
| `tiff-3.7.0` | Inc. Error | 0 | 0 | 0 |
| PPLOC: 172,842 | Confusion | 3 | 0 | - |
| | Other | 0 | 3 | 56 |
| Dead code | Trivial | 1 | 0 | - |
| | Macro | 3 | 1 | - |
| False positives | Boundary FP | 2 | - | - |

Table 7.3: Warnings from Other Open Source Packages

| Benchmark | Category | Univ. Reach. | Saturn | Clang |
|---|---|---|---|---|
| SAMATE | Inc. Error | 6+16 | 6+0 | 6 |
| | Boundary | 836 | - | - |
| | Confusion | 1 | 0 | 0 |
| | Other | 0 | 20 | 377 |
| | Configuration | 2 | 0 | 0 |

Table 7.4: Warnings from SAMATE Benchmarks

excels, and we see that Curate can detect the majority of the buffer overflow defects. Although the buffers in these test cases are all of constant, known sizes, universal reachability analysis is able to detect overflows in more complex cases as long as the relationship between program variables and buffer sizes is known (such as in the example of Sec. 5.1).

In addition to the buffer overflows detected by boundary condition specialization, Curate is also able to discover a number of cases of null pointer dereferencing and freeing already-freed pointers. The null pointer dereferences are detected by both Saturn and the Clang Analyzer.

Finally, the Clang Analyzer generates a large number of warnings outside of the realm of inconsistency analysis. Most of these indicate values written and then never read.

Overall, these results suggest that boundary condition specialization, and potentially other variants of universal reachability analysis, can detect interesting bugs, even though it leads to large numbers of false positives in some cases. Therefore, it likely makes sense to include it as an optional component of a universal reachability analysis implementation.

## 7.9 Performance

Along with defect detection rates, we also measured the time and space taken to perform various aspects of universal reachability analysis. First, we measured the total time taken, per benchmark, for each of the three tools. We also instrumented Curate to record the time taken to analyze each function, as well as the sizes of the

| Benchmark | Functions | Analysis Time (H:M:S)/Timeouts | | | | | |
|---|---|---|---|---|---|---|---|
| | | Curate | | Saturn | | Clang | |
| MPlayer-1.0pre8 | 7687 | 10:46:55 | 258 | 8:49:38 | 170 | 22:34 | 0 |
| openssh-4.3p2 | 1020 | 14:41 | 4 | 45:23 | 14 | 1:31 | 0 |
| openssl-0.9.8b | 4591 | 1:16:01 | 15 | 10:46:39 | 191 | 3:44 | 0 |
| samba-3.0.23b | 6814 | 1:44:51 | 12 | 33:59:38 | 738 | 10:23 | 0 |
| sendmail-8.13.8 | 810 | 1:12:59 | 26 | 3:59:20 | 88 | 5:22 | 0 |
| bc-1.06 | 251 | 3:56 | 1 | 23:11 | 12 | 0:21 | 0 |
| js-1.70 | 1914 | 1:13:46 | 25 | 5:00:36 | 137 | 2:59 | 0 |
| ncompress-4.2.4 | 44 | 2:29 | 0 | 0:39 | 0 | 0:03 | 0 |
| pcc-0.9.9 | 473 | 18:45 | 7 | 46:35 | 20 | 0:57 | 0 |
| squid-2.3.STABLE1 | 1508 | 10:34 | 1 | 1:57:25 | 50 | 1:43 | 0 |
| tiff-3.7.0 | 571 | 13:14 | 4 | 28:48 | 7 | 1:02 | 0 |
| SAMATE | 1731 | 4:57 | 0 | 24:29 | 0 | 0:56 | 0 |
| Total | 27414 | 17:23:08 | 353 | 67:22:21 | 1427 | 51:35 | 0 |

Table 7.5: Overall Analysis Time

original C statements, the resulting control flow graphs (CFGs), and finally the passive versions of these graphs.

### 7.9.1 Overall Time

We measured the elapsed time for all three analysis tools, and show the results in Fig. 7.5. The times for Saturn and Curate include only analysis time, and are calculated using timing routines built in to the tools. Parsing time (for Curate) and database construction time (for Saturn) are not included. Similarly, the time to do exit function analysis, the results of which are shared between the two tools, is not included. The time taken by the Clang Analyzer, however, is elapsed time as reported by the Unix `time` command.

We see that Curate usually takes less time than Saturn, and times out less often. The Clang Analyzer is at least an order of magnitude faster than either of the other tools in every case except the SAMATE benchmark, for which Curate takes only 5 times as long.

These numbers align closely with the engineering choices and sophistication of analysis that make up each tool. Saturn performs inter-procedural analysis and uses

a pure satisfiability (SAT) solver to check for inconsistencies. Curate, on the other hand, performs an intra-procedural analysis, and uses an satisfiability modulo theories (SMT) solver, which can solve certain some queries more efficiently. Finally, the Clang Analyzer performs a well-engineered but coarse symbolic execution that is very effective at finding simple bugs that fit within the realms of pattern matching or dataflow analysis. This makes Clang appropriate for interactive use during development, but limits it to somewhat shallow defects.

Ultimately, both Curate and Saturn are practical for batch (perhaps overnight) analysis of fairly large projects (millions of PPLOC), and discover relatively deep defects. The Clang Analyzer, on the other hand, is appropriate for interactive use on projects of essentially any size, but discovers shallower defects.

### 7.9.2 Effects of Slicing

Universal reachability analysis of a statement requires analysis of a number of slices, each of which is similar in size and structure to the original statement. The number of slices required for a given statement is a potential impediment to scalability.

Fortunately, the similarity of each slice to the original statement means that an incremental decision procedure might offer significant speedup when analyzing statements with large numbers of slices. To determine the necessity of an incremental approach, we measured the number of slices generated from each function body in all of the benchmarks described above. The distribution of slice counts appears in Fig. 7.1, plotted alongside the inverse cubic function $60000x^{-3}$. The first bar, representing counts between 1 and 10, makes up 88% of the total. In the worst case, we saw one function with 460 slices. These numbers suggest that an incremental approach would be helpful, but is not crucial.

To complement the slice count, we also measured the total time taken to perform universal reachability analysis on each function body. Curate ran with a time limit of 120 seconds. The analysis failed to complete on some functions within this time limit, as shown previously in Fig. 7.5. The measured time includes the total time to analyze all of the slices. The distribution of analysis times, in seconds, appears in Fig. 7.2. In this histogram, the first bar represents 98% of function bodies, all of which

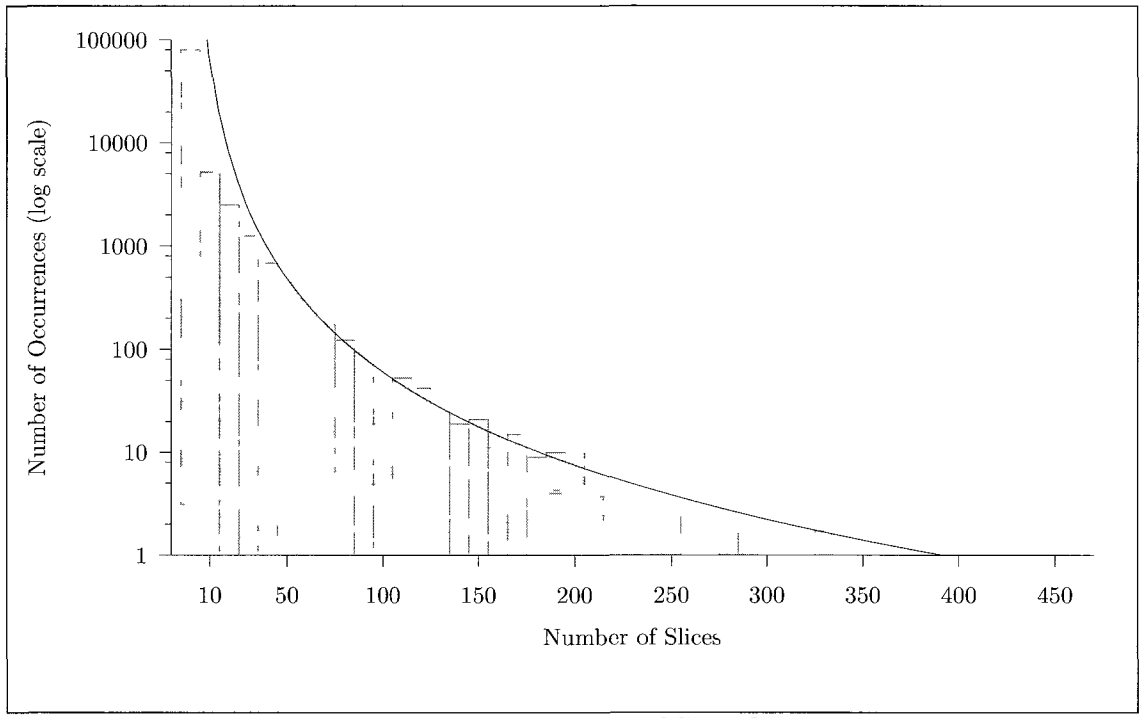


**Figure 7.1:** Distribution of Slice Counts

took less than one second to analyze. Although, unlike the slice counts, running times do not fit a polynomial function as closely, they show an even sharper decline. We show them compared to the function $2000x^{-1.8}$ for reference. Again, we see support for the conclusion that incremental solving would be beneficial but is not essential.

### 7.9.3 Intermediate Language Size

Along with analysis time, memory use is an important factor in the practicality and scalability of an automated program analysis. Although all tools successfully processed all benchmarks on the final test system, with 8GB of RAM, earlier experiments on machines with less available memory did not always complete without running out of memory.

To determine what factors affected memory use, we calculated the relationship between the sizes of source abstract syntax trees (ASTs) and the sizes of the resulting control flow graphs, as well as the relationship between the sizes of imperative and passive versions of the internal representation. Fig. 7.3 compares the number of nodes

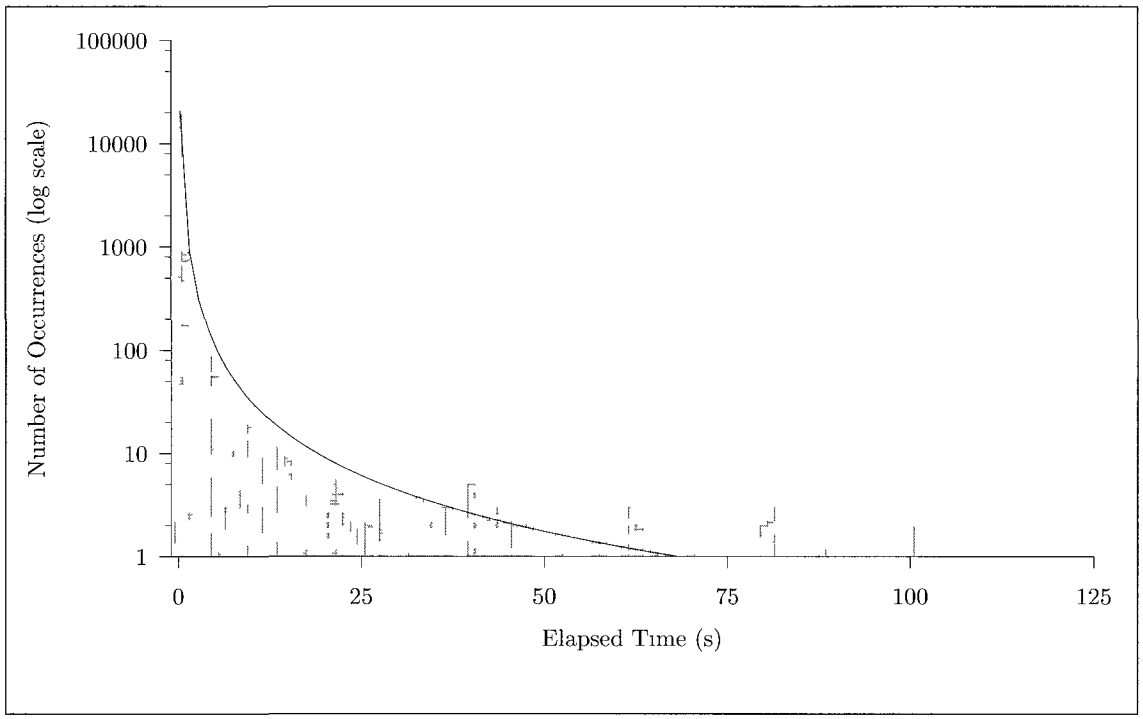


**Figure 7.2:** Distribution of Function Analysis Times

in the AST representing a function body with the number of vertices in the resulting control flow graph. Fig. 7.4 compares the number of vertices in the imperative control flow graph with the number after translation to passive form.

AST nodes include more fine-grained detail than control flow graph vertices, as subtrees exist for every sub-expression in the program. Therefore, the ASTs almost always have a larger number of nodes. However, we can see that the relationship between AST and control flow graph sizes is roughly linear. The number of vertices in the resulting graph is at most approximately the same as the number of nodes in the original AST, and is rarely less than one quarter of this number.

When transforming from imperative to passive representation, the number of graph vertices never decreases. While in this case, the relationship is less clearly linear, the vast majority of cases involve less than a four-fold increase. Grigore *et al.* have shown that the passivation algorithm we use adds $O(|E|)$ vertices for copy instructions [52].

Finally, we measured the distribution of overall graph sizes, as shown in Fig. 7.5 alongside the inverse cubic function $40000x^{-3}$. The first bar represents vertex counts

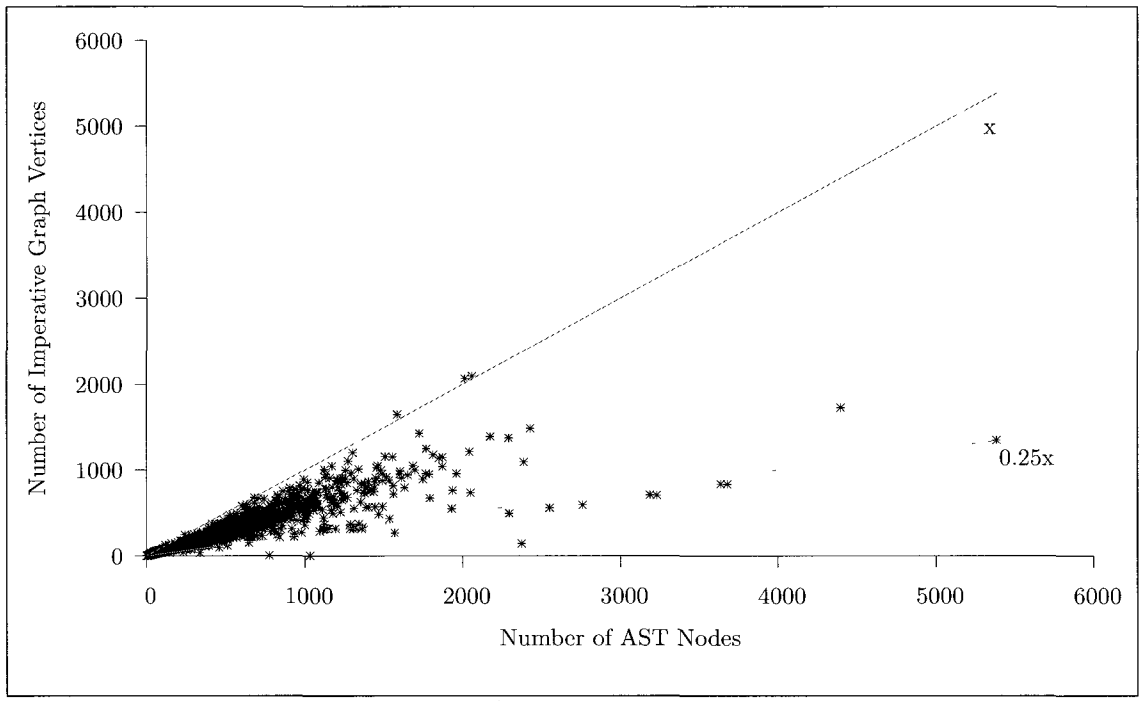


**Figure 7.3:** Relative Sizes of AST and Imperative CFG Representations

from zero to 99, covering 92% of the function bodies in our benchmark collection. The largest passive graph we observed had 6631 vertices, and only 0.2% of cases contained more than 1000 vertices.

The observed vertex counts suggest that our experience with memory exhaustion in a few cases would likely be avoidable with sufficient engineering effort. In an efficient implementation, many statement vertices could be shared between imperative and passive graphs, the original AST could be discarded once the control flow graphs were constructed, and the generated verification and failure conditions could share subexpressions with the labels on the graph vertices.

## 7.10 Discussion

When looking for items that fell into our definition of inconsistency, Curate found all the bugs that Saturn found except for those obscured by our translation of short circuit operators. Saturn finds some additional bugs that Curate does not because they fall outside of our definition of inconsistency, but Curate, in turn, finds bugs that

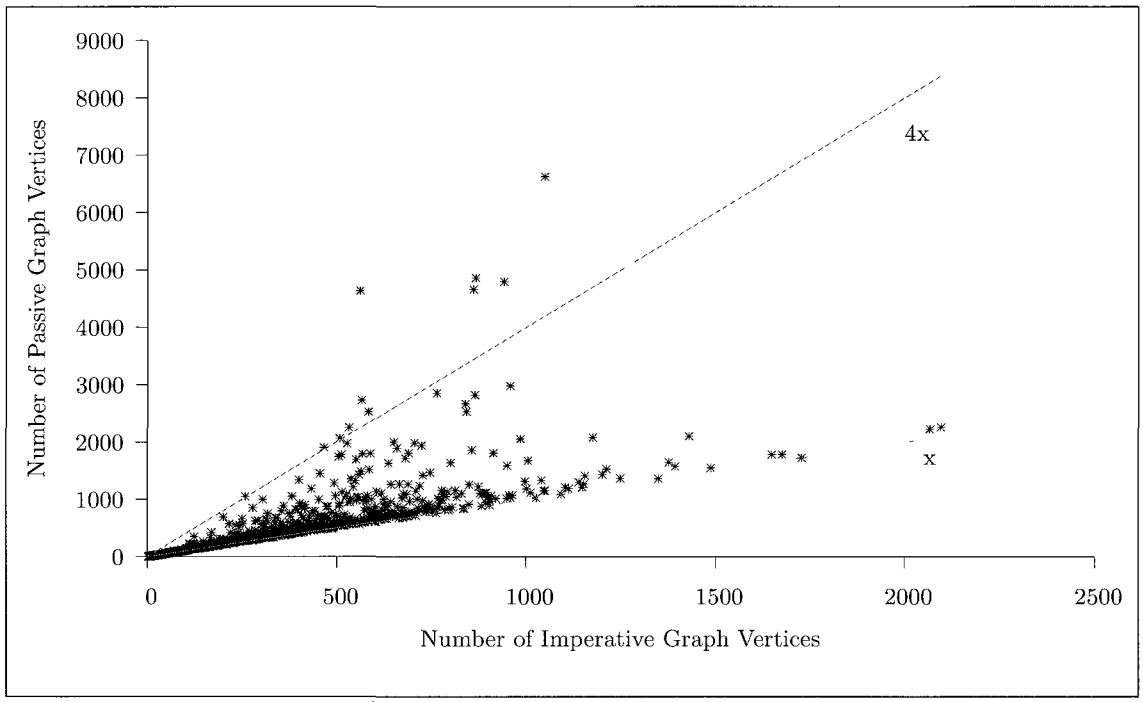


**Figure 7.4:** Relative Sizes of Imperative and Passive CFG Representations

Saturn cannot thanks to Curate's use of a more expressive logic.

While some of the bugs discovered by Saturn's null pointer analysis are not considered inconsistencies by our definition, in some cases they still represented significant problems. Most often they consisted of a pointer dereference guarded by a null check in one of two mutually-exclusive branches but not the other. In many cases, this pattern is intentional, and generally does not lead to null pointer dereferences. However, in some cases, they took a more general form, and some of the more interesting bugs detected in a few of the benchmarks fell into this category.

Broadly, the inconsistencies that Saturn can detect and Curate cannot take the following form:

```
if (c1) { if (x) y = *x; }
if (c3) { y = *x; }
```

In universal reachability analysis, even on the slice where the `if` (x) ... follows the `else` branch, there still exist inputs that allow termination, namely any where c3 is false. Therefore, this "parallel" inconsistency detection is a worthwhile complement to

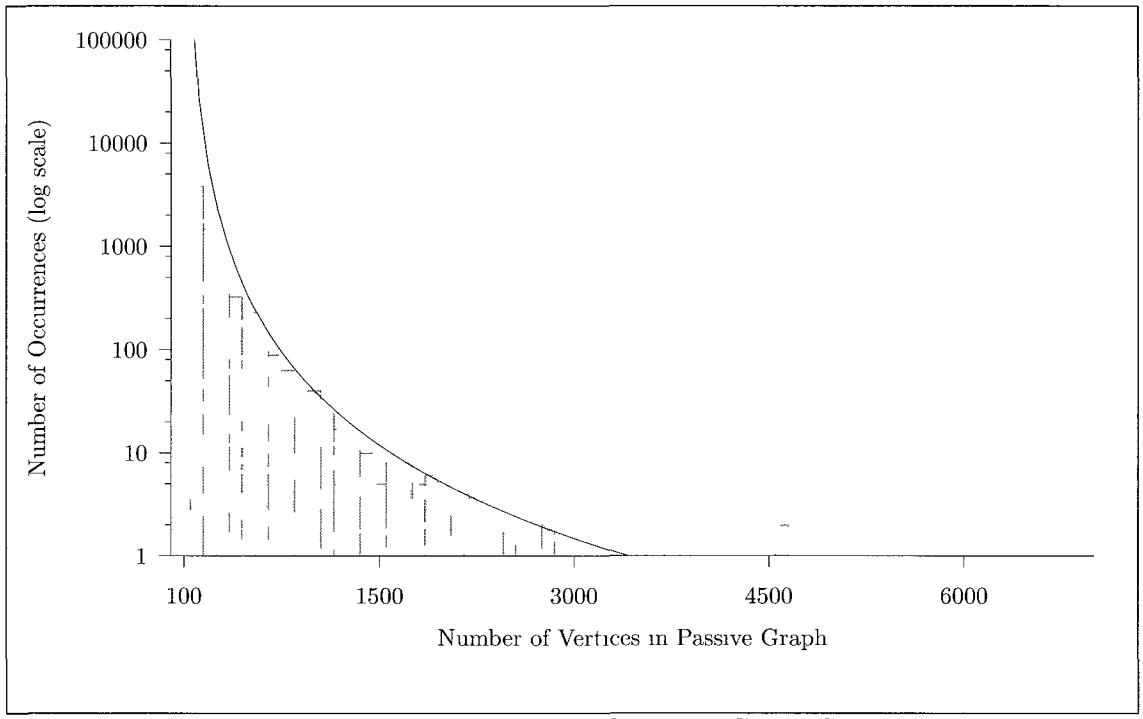


**Figure 7.5:** Distribution of Passive Graph Sizes

our approach.

Curate generally produced fewer warnings than the Clang Analyzer. For very large programs, however, universal reachability analysis may be practical only on targeted areas, because of the number of warnings generated, unless it is used throughout the entire development and inconsistencies are addressed as they occur. The same conclusion applies to most defect-detection tools, however.

Curate and Saturn's null pointer analysis are both significantly slower than the Clang Analyzer. Curate was still able to process all of our chosen benchmarks in less than a day on a modern workstation, however, and Saturn required a little under three days.

We found that boundary condition specialization helped the analysis discover interesting bugs — perhaps the most interesting in the entire collection of discovered defects — but at the cost of a high rate of false positives, especially on the MPlayer benchmark.

Macros were the primary cause of benign inconsistencies. In a production-

quality tool, it would be valuable for the parser to keep track of information about macro expansion, so that later analyses could treat many uses of macros as if they were function calls. As a programming practice, our experience lends weight to the philosophy that inline functions are preferable to macros for small, lightweight abstractions.

Ultimately, universal reachability analysis appears to be a good companion to existing approaches, and a technique that may be worth integrating into a more wide-ranging tool in the future. Boundary condition specialization should be an option in such a tool, but probably off by default.

We found that the number of slices in any one function can be large, but not prohibitively so. Incremental solving would probably improve efficiency, but is not crucial. We also confirmed earlier results that passivation does not prohibitively increase the size of a program, and found that it rarely increased the instruction count of the program by more than four times.

In summary, Saturn's analysis can detect a few defect types that Curate cannot, and could also probably be generalized to more expressive logics. The results of Dillig *et al.* show that inter-procedural analysis is feasible for the null analysis Saturn performs. It is unclear whether it would continue to be feasible with a more expressive logic.

The next tool we evaluated, The Clang Analyzer, is an ideal tool for interactive use, along with similar tools such as FindBugs and Splint. With increases in computing power, or more sophisticated implementation, inconsistency analysis may become appropriate for an interactive setting as well, especially if used incrementally, limited to recently-modified code.

Finally, we found that universal reachability analysis has advantages when compared with other tools. It is likely to be even more powerful if libraries are annotated with detailed specifications. It has the benefit of being able to handle any logic that can be used with weakest preconditions, though it is not clear how expensive inter-procedural analysis would be.

# Chapter 8

# Related Work

The work we present builds on a large body of earlier research. Some of the research forms a direct foundation for universal reachability analysis. Other work is complementary and could be beneficially combined with our analysis.

## 8.1 Inconsistency Analysis

The work presented in this dissertation is inspired by previous work on semantic inconsistencies. Much of the current work in this area can be said to build on the 2001 paper, *Bugs as Deviant Behavior: a General Approach to Inferring Errors in Systems Code* [39], by Engler *et al.*, which discusses contradictions between the beliefs implied by particular constructs in source code. Engler co-founded Coverity, a company focused on building tools for static detection of software defects. He has claimed that every static analysis system he has built since then has made use of the technique of inferring programmer beliefs [38]. Most of these inconsistency-detection tools used symbolic execution as a core analysis technique [17, 96, 16].

One of the key works building on Engler's research is *Static Error Detection Using Semantic Inconsistency Inference*, by Dillig, *et al.* [37]. Their analysis operates on the same overall principle as those of Engler *et al.*, namely searching for cases where code indicates contradictory programmer beliefs. However, the analysis of Dillig *et al.* takes an approach reminiscent of type inference [69] or constraint-based analysis [4], rather than symbolic execution.

The class of bugs detected by our analysis overlaps significantly with the work of Dillig *et al.* Specifically, the inconsistencies discovered by their analysis can be divided into sequential and non-sequential inconsistencies. A sequential inconsistency is one in which two program points that can execute in sequence along a single path make conflicting assumptions. For example, consider the following fragment of C code:

```
if(x) y = *x;
...
z = *x;
```

The first assignment suggests that x is allowed to be null. However, the second assignment will only succeed if x is non-null. A non-sequential inconsistency, on the other hand, is one in which two program points that may not both execute along the same path make conflicting assumptions. For example:

```
if(c1) {
    ...
    if(x) y = *x;
    ...
}
if(c2) {
    ...
    z = *x;
    ...
}
```

Here, the fact that the dereference is guarded by a null check in one case but not the other suggests an error. However, consider the slice in which the `else` branch of the null check executes. It is still possible for this slice to terminate, because it is possible that c2 is false, and therefore the second dereference never occurs.

Any sequential inconsistency discovered by the analysis of Dillig *et al.* will also be discovered by our analysis. Non-sequential inconsistencies will be discovered by their analysis but not ours. Sequential inconsistencies that involve richer expressions than equality constraints will be discovered by our analysis but not theirs. This latter

class of inconsistencies includes the example described in Ch. 1. Their analysis could, however, be extended to handle more complex constraints.

Note that sequential inconsistencies will either lead to assertion failures or involve dead code. Parallel inconsistencies may be neither. They suggest the possibility of conflicting programmer beliefs, but in our experience led to more false positives than true bugs, though some did indicate significant bugs. Many examples of C code use a single variable to store various types of information over time, and use other conditions to determine what invariants that variable should satisfy at any particular program point. The analysis of Dillig *et al.* addresses this problem with a correlation analysis that attempts to determine whether conditions on one variable affect the beliefs about another. Our analysis, instead, does not identify parallel inconsistencies.

## 8.2 Static and Dynamic Contract Checking

Many programming languages and external analysis tools have adopted the use of one of a variety of contract systems, like the pre- and postcondition annotations used earlier in this dissertation.

The Eiffel programming language [68] emphasizes the notion of Design by Contract, in which specifications written in Eiffel code accompany the implementation. This technique allows for very expressive specifications, and has also been adopted by a number of other languages [66, 82, 86]. However, these systems, as implemented, execute all of the specification code dynamically, when method calls occur. Therefore, they have low coverage; may detect bugs late in development, or after deployment; and can lead to reduced runtime performance. Dynamic contract checking has the advantage, on the other hand, of avoiding false positives. Any contract violation discovered during a dynamic contract check is accompanied by a specific runtime state that is guaranteed to disagree with the contract.

Systems such as ESC/Java [47] and Spec# [12] also adopt the technique of adding contracts to implementation code, but try to statically verify that the contracts hold, rather than depending on dynamic checks. The static verification involves using an automated theorem prover to attempt to discharge verification conditions derived from the combination of source code and specifications. The warnings these tools generate

when they fail to discharge verification conditions fall into one of three categories:

**Contract errors.** Missing or incorrect contracts can lead to warnings even when the code they describe is correct. If a function has a postcondition annotation but a missing or overly weak precondition annotation, for instance, it may be impossible to prove that the postcondition is satisfied, even if it is in practice. Static contract checkers will also warn if an annotation directly contradicts the code. Universal reachability analysis, on the other hand, will not warn in the presence of missing or overly-weak contracts, but will warn in the case of direct contradictions.

**Code errors.** If contracts are present and correct, the code may still be incorrect. Both existing static contract checkers and universal reachability analysis will detect code errors.

**Tool limitations.** A tool may have an incorrect model of the underlying language, leading to warnings about problems that occur in the model, but not in the actual code. Alternatively, the impossibility of creating a complete decision procedure for most logics of interest means that an automated prover may fail to prove a verification condition that is in fact valid. The former issue applies equally to universal reachability analysis. The latter does not, because we warn only in cases where we can successfully prove that a non-terminability failure condition is valid (assuming our decision procedure is sound but incomplete).

The Krakatoa system [64] analyzes Java programs annotated with the Java Modeling Language (JML, also used by later versions of ESC/Java), but can use an interactive proof assistant (such as Coq [84]), rather than an automated prover, to discharge verification conditions. Frama-C is a similar tool for C programs [1].

One benefit of using an interactive prover is that tool limitations are diminished. Many proofs that an automated prover would fail to discover can be manually constructed by a human user. However, completing a correctness proof using an interactive prover requires a great deal of manual annotation and theorem proving effort. In addition, tool limitations due to imprecise or inaccurate modeling of language semantics can still lead to either false positives or false negatives.

The techniques used in systems such as ESC/Java, Spec#, Krakatoa, and Frama-C could be applied to languages such as Eiffel in which contracts are typically dynamically checked. Other techniques, such as universal reachability analysis and symbolic execution, can also be used to check contracts, as in the inconsistency-based tools mentioned in Sec. 8.1, and in the tools that combine static and dynamic analysis, described in Sec. 8.7.

## 8.3 Model Checking

Model checking is a popular technique for both verification and bug finding. It works by building a graph (which may be implicit or explicit) of all possible states in the system and the transitions between them. Given this graph, it is possible to check that, for instance, every reachable state satisfies some safety condition, or conversely that no state that satisfies an error condition is reachable.

For small systems, especially hardware designs, model checking is very effective. For larger systems, the number of possible states in the system can become prohibitively large (the "state space explosion" problem). Several techniques exist to address this problem:

- Symbolic representations of program states in forms such as Binary Decision Diagrams (BDDs) [15] can reduce the amount of information the analysis needs to keep in memory.
- Counterexample-guided Abstraction Refinement (CEGAR) [55, 21, 9] makes it possible to model-check a coarse approximation of a system. If the property of interest holds on the approximation, it is guaranteed to hold on the real system. If a counterexample is found, it can be used to refine the abstraction. This process continues until it can be shown that the counter-example applies to the original system.

Even with these techniques, however, model checking has not managed to scale up to programs of the size we address in our experiments. And even for programs that model checkers can handle, Engler has observed that traditional static analysis techniques often find more bugs [40].

## 8.4 Dependent Type Systems

In dependent type systems, types can be parameterized by terms. Dependent types provide more flexible and expressive program specifications than non-dependent types, and can express similar properties to those possible with contract systems. Like contract checking, dependent type checking becomes undecidable when the types become sufficiently expressive.

Early work on making dependent types practical [95, 94] focuses on restricting the language of terms allowed inside types in such a way that the type system remains decidable. The result is a more expressive specification language than provided by classical type systems, while remaining decidable. However, these types are often too restrictive to express many desirable properties.

The Cayenne language [7] uses an unrestricted dependent type system, in that types can be parameterized by arbitrary terms, giving it significant expressive power. It achieves this by largely ignoring the problem of undecidability. Cayenne's type checker attempts to prove that programs are type correct, but simply rejects them if the type-checking takes longer than a specified time. We would instead accept such a program without producing any warnings.

Hoare Type Theory [71] combines Hoare logic (using potentially undecidable theories) with Hindley-Milner-style type systems, and later incarnations extend its power for reasoning about heap structures by integrating separation logic as well [72]. Due to the potential undecidability of the type system, the primary implementation of Hoare Type Theory, YNot [73], uses an interactive theorem prover (Coq) to discharge the proof obligations that arise during type checking. Because of the strong correspondence between inconsistency analysis and type checking, it would be interesting to recast universal reachability analysis within the framework of Hoare Type Theory.

All of these systems require extensive type annotations as soon as expressive dependent types are used anywhere. Some other systems attempt to reduce the annotation burden by resorting to dynamic checks in some cases.

Ou *et al.*, describe a type system in which some components are simply-typed and others are dependently-typed [81]. It uses runtime checks to ensure type safety at the interface between the simply-typed and dependently-typed components. This

allows the programmer to control the degree to which they wish to use dependent type annotations. However, their system will reject a program that, for instance, passes a dependently-typed argument to a dependently-typed function if it cannot prove type safety. Universal reachability analysis would reject the program only if it could prove that type correctness would be violated under specific conditions. Therefore, we allow for an even smoother continuum of annotations.

Hybrid type checking [44, 53, 45] also uses an undecidable dependent type system, and has a similar philosophy to universal reachability analysis in that it rejects a program only if it can prove the program violates its specification. If the type checker cannot prove type soundness, but cannot disprove it either, it resorts to dynamic checking. The use of dynamic checks in this case has two advantages: it avoids the problem of undecidability in the type system, and it allows a gradual adoption of dependent type annotations.

Gradual type systems take a similar approach, focusing on the latter benefit of allowing any degree of annotation along a spectrum, from completely dynamic to completely static [89, 88, 90]. In gradual type systems, type checking has typically been decidable, but some components may be untyped (or marked implicitly with the type Dynamic). Interfaces between code that has statically-known types and code that uses dynamic types require runtime casting operations that may fail.

## 8.5 Specification Inference

Several researchers have focused on the problem of inferring specifications, which can then be used to aid later analysis such as static contract checking or universal reachability.

The Houdini tool [46] for ESC/Java attempts to generate correct specifications by producing a number of candidate annotations and using ESC/Java to verify or refute each candidate. Yannick Moy describes another technique for statically inferring preconditions sufficient to guarantee crash freedom [70] that uses a combination of weakest precondition analysis and abstract interpretation.

Other approaches have used dynamic analysis, or combinations of static and dynamic analysis to derive possible specifications for procedures or data structures. The

Daikon tool [41] looks for patterns in program execution traces in an attempt to discover likely invariants. Nimmer and Ernst use the output of Daikon to generate ESC/Java annotations [77]. Csallner, Tillmann, and Smaragdakis describe a technique that extends the approach of Daikon using by symbolic execution alongside concrete execution, producing invariants that human users are more likely to consider relevant [32]. Ammons, Bodik, and Larus describe an alternative technique that uses machine learning to derive state machines that capture the observed behavior of software [6].

Any of these techniques could be a useful way to improve the power of universal reachability analysis when applied to previously unannotated or partially-annotated programs.

## 8.6 Other Static Bug Finding Approaches

Many tools have been developed to automate the search for software bugs in programs without contracts or specifications, using a variety of sound semantic analyses, unsound heuristics and pattern-matching approaches. One of the earliest tools, based on a collection of largely heuristic approaches, was the C analyzer Lint [61]. Lint inspired the later LCLint tool, which was then renamed Splint [42].

More recently, the FindBugs tool has applied a variety of primarily pattern-matching techniques to find bugs in Java programs [58]. FindBugs includes a component to check for null pointer dereferences that uses dataflow analysis plus a collection of heuristics to handle common code patterns. These heuristics build in some of the same ideas present in the tools that use inconsistency-based techniques, and FindBugs is therefore able to discover some of the same bugs that Curate and Saturn's null pointer analysis can detect. However, because FindBugs works on Java programs, and Curate analyzes C, we are not able to directly compare the tools.

Another recent bug-finding tool is the Clang Analyzer [83], part of the LLVM Compiler Infrastructure Project and included as part of our experimental analysis. The Clang Analyzer performs symbolic execution over the source program and allows a collection of pattern-recognizing plugins to execute at various points during the simulation. It includes a null pointer analysis, which found a few of the potential null pointer dereferences detected by Saturn and Curate, as described in Ch. 7, though it missed the

majority of the bugs detected by either of the other tools.

## 8.7 Other Combinations of Static and Dynamic Analysis

Because our analysis focuses primarily on determining statically whether various dynamic checks will be certain to fail at runtime, it has some similarities with other defect-detection techniques that combine static and dynamic analysis.

In one of the earlier combinations of static and dynamic type systems, Abadi *et al.*, describe a simple static type system with a special type Dynamic [2]. Values of type Dynamic are created by adding a tag to a value of some specific type, and inspected using a type-safe construct that performs different operations on differently-tagged values.

A short opinion paper by Bracha suggests the notion of *pluggable type systems* [14]. His thesis is that current languages connect the operational semantics and type systems of programming languages too tightly, making the use of those languages inflexible. Instead, he proposes that we build our languages purely from descriptions of syntax and operational semantics, and add type systems on in a modular way. Thus, a single language could use type systems of various complexities (ranging from purely dynamic typing, to sophisticated dependent types) depending on the balance between the need for assurances of correctness, and the desire for flexibility and rapid prototyping. Universal reachability analysis could be seen from this angle: not required by the language definition, but a useful supplemental property.

Also the realm of types, soft typing systems [18, 5, 65] typically take a conflict-based approach to type checking that shares many characteristics with universal reachability analysis. Such systems usually attempt to infer types, but may not always be able to ensure that the program satisfies the inferred types. In this case, rather than rejecting the program, they insert dynamic checks to ensure type correctness at runtime. As a result, they only warn about programs in which they can guarantee that there is a type conflict.

In a similar vein, CCured [74] translates C programs into memory-safe equivalents by inserting sufficient runtime checks to guarantee that all pointer dereferences and array accesses are safe. If it does not insert any runtime checks, the program is statically guaranteed to be memory safe. Otherwise, the program will terminate on any

unsafe access, instead of allowing the program to proceed in a corrupted state. Our analysis only warns if it can prove that an unsafe access is possible, on the other hand, and does not modify the original program.

A number of systems have also used static analysis techniques to determine potential error-causing inputs, and concrete execution to check whether these inputs do indeed lead to errors. JCrasher [29] and QuickCheck [20] simply execute a program with random inputs, and check for crashes or violations of specified properties.

Directed Automated Random Testing (DART) [50] looks for runtime errors by using a test harness that executes a portion of code with random inputs, observes how the code executes, and uses its observations to generate new random inputs (using a constraint solver) that will exercise other paths through the code. The focus on ensuring that each path can potentially terminate successfully (in their case, on a random value) is similar to ours. Our approach is purely static, but will in general consider a smaller fraction of the possible paths in the program. A more recent variation on DART uses the term *concolic testing* to refer the synthesis of concrete and symbolic execution [87].

The Check'n'Crash [30] system combines JCrasher and ESC/Java, using ESC/Java's warnings to generate conditions under which the program may crash, and JCrasher to generate concrete test inputs derived from these conditions. If the tests actually cause a crash, then they indicate real bugs. Finally, DSD-Crasher [31] adds Daikon's invariant detection to the beginning of the Check'n'Crash pipeline, using the inferred invariants to generate potential ESC/Java annotations. These approaches provide an alternative method for dealing with initially-unannotated programs.

Another combination of static and dynamic analysis, due to Tomb, Visser, and Brat [92], uses symbolic execution to derive conditions under which a runtime error may occur, and constraint solving to derive concrete inputs that may trigger the error. It then runs the program on those inputs, and warns only if the concrete inputs lead to a runtime exception. This system has many of the same defect-detection properties as Check'n'Crash. The main difference is that symbolic execution discovers somewhat different potential errors than ESC/Java's Hoare-logic-based approach.

# Chapter 9

# Future Work

The theory and experimentation reported in this dissertation suggest various lines of potential future work. We have considered various extensions to the core theoretical foundations, as well as more thorough implementation and experimentation, and the use of alternative logics in an implementation.

## 9.1 Theoretical Extensions

In Sec. 3.2 we describe a slicing function based directly on the branches that exist in the program, with no additional constraints. We then show how each of these slices corresponds to a subset of the entire space of initial states for the program in Sec. 5.1, and go on to describe an technique for further constraining that state space based on the boundary cases of inequality predicates. However, we have no reason to think that this specialization is the only interesting possibility. Perhaps other specializations would provide as much increased bug-finding power as boundary condition slicing without as many false positives.

On another axis of the design space, it would be interesting to consider how the basic ideas we present might apply to programs written using different programming paradigms. Our current approach is most appropriate for imperative programming languages. Other similar approaches may be relevant for other paradigms.

In many cases, the failure conditions generate by our analysis are too precise. That is to say, the bugs that we detect could be demonstrated by much simpler

predicates. It is possible that the techniques of counterexample-guided abstraction refinement that have had success in the model-checking domain [21] could be applicable in the setting of analyses based on Hoare logic.

Finally, it could be interesting to investigate using the techniques of the Terminator tool [25, 26] to analyze the terminability of slices.

## 9.2 More Experimental Validation

We have presented initial experimental results in Ch. 7 to validate the technique of universal reachability analysis. However, our implementation, Curate, has a variety of weaknesses, and a more thorough implementation could be more powerful and practically applicable.

### 9.2.1 Practical Interprocedural Analysis

Curate takes a very conservative approach to procedure calls, by assuming that they may write to any location in the heap. This approach is a correct over-approximation of program behavior, but potentially very imprecise. The most appropriate treatment of procedure calls remains an open question. As we describe in Sec. 3.5, inlining the body of a target function at its call site leads to many uninteresting inconsistencies.

If all procedures happen to be annotated with specifications that describe their complete behavior, the weakest precondition operation described in Sec. 3.3 would provide a precise and potentially efficient interpretation of procedure calls.

In the absence of annotations, however, some inference of procedure behavior is still possible. One alternative would be to infer the effects of procedures, as a pre-processing phase, and use the inferred effects to restrict the scope of the havoc instruction we insert in place of procedure calls. Alternatively, we could use weakest precondition analysis to infer a specification for each procedure. In the presence of recursion, it would be necessary to introduce some sort of predicate widening operation to guarantee termination, but this could also help in the analysis of loops, by allowing us to infer loop invariants and therefore more closely approximate the least-fixpoint semantics of loops.

It remains to be seen how more specification inference, either at the level of effect inference or at the level of precondition inference, would affect either the defect-detection power or the efficiency of universal reachability analysis.

### 9.2.2 Wider Range of Library Specifications

Curate provides specifications for some of the core expressions in C, as well as a few library functions, essentially providing a foundation for the detection of memory safety violations. Specifications exist for pointer dereferencing, array indexing, division, and allocation and release of memory. Many more standard library functions have interesting preconditions, and it may be possible to use them to detect more subtle and interesting bugs.

### 9.2.3 More Precise Implementations

Many aspects of Curate are somewhat imprecise. In particular, we approximate the semantics of the following operations:

**Bit-level operations** We treat operations on individual bits as uninterpreted functions.

**Floating point** We treat operations on floating-point values as uninterpreted functions.

**Signedness, overflow, and underflow** We treat all integral values as mathematical integers, which may be positive or negative, never overflow, and never underflow.

**Casting** Because of the simple semantics for integers, and the lack of floating-point support, casts are ignored.

**Procedure calls** As mentioned above, procedure calls obscure all knowledge of the contents of the heap.

All of these could be improved using known techniques, with sufficient engineering effort. More precise treatment of these operations could help reduce the false-positive rate, and could lead to detection of additional bugs. It also may be valuable to use a more sophisticated memory model, such as that from seL4 [62] or VCC [23].

## 9.3 Alternative Logics

Our theoretical presentation is largely agnostic, by design, to the details of the logic used in an implementation. In Curate, we chose a quantifier-free subset of first-order logic, including theories of equality, uninterpreted functions, arrays, and integer arithmetic, because that combination is widely supported by existing SMT solvers, such as the Yices solver we chose, and because it corresponds fairly closely to the range of specifications that can be directly expressed using unmodified C expressions.

However, given an appropriate decision procedure, an alternative logic could be either more powerful or more efficient. One important additional capability would be more complete support for reasoning about structures in the heap. Traditional verification conditions based on weakest preconditions describe the heap in ways that leak implementation information over abstraction boundaries, hampering compositionality. Separation logic provides and elegant solution to the problem of local, compositional heap reasoning [80]. Unfortunately, an algorithm to compute weakest preconditions over the full range of separation logic does not yet exist, nor do decision procedures exist for the entire logic (though they do exist for subsets [13]).

Matching logic is another alternative, which includes more concrete configurations than standard Hoare logic, allowing for a more natural treatment of heap structures [85].

Finally, even within the logic we have chosen, a wide range of decision procedure implementations exist. Yices performs well, but still sometimes has trouble with the verification and failure conditions we generate. On programs that take significant time to analyze, the decision procedure is almost always the bottleneck. Alternative provers such as Z3 [33], Simplify [34], and Alt-Ergo [24] are promising possibilities that have performed better under certain circumstances.

## 9.4 Usability

The most straightforward error messages for universal reachability violations can sometimes be insufficient to clearly communicate the source of the problem. When a slice is not terminable, the most readily available information is the identity of the

slice: the branch it forces, and the condition associated with that branch. In simple cases, this information is sufficient to diagnose what error may occur. However, for more complex programs, the source of the fault can remain obscure.

Performing further analysis on a non-terminable slice to determine what assertion may fail, and what other assertions or assumptions it conflicts with, could make the results of universal reachability analysis more readily useful. This further analysis could also have the benefit of improving the identification of false positives due to imprecision, or uninteresting inconsistencies due to dead code.

# Chapter 10

# Summary

In the last ten years, research has shown that automated tools for detecting inconsistencies in software can reveal serious bugs. In this dissertation, we have argued that axiomatic semantics can form the basis of an elegant and extensible technique for inconsistency detection that generalizes previous approaches. Our approach retains the benefits of previous inconsistency-detection techniques, and brings advantages of its own.

The idea of *inconsistency* as a source of program defects has likely been in the minds of programmers for decades. A program is inconsistent if it makes assumptions, either explicitly or implicitly, that contradict assumptions made elsewhere. Inconsistencies almost certainly indicate mistakes of some sort, and automated tools to search for them have become popular in the last decade, in part because of the work of Dawson Engler and his colleagues and students at Stanford. Their work culminated in several open source [39, 16] and commercial tools [27]. Later, another group at Stanford, led by Alex Aiken, formalized a particular approach to inconsistency analysis and used it to implement a prototype detector for potential null pointer dereferences within the Saturn program analysis infrastructure [37].

An automated defect-detection tool with the goal of searching for inconsistencies has several advantages, regardless of the specific details of the analysis:

- It should not, in theory, lead to significant numbers of false positives. In practice, some inconsistencies are inevitably present in correct code. These are true inconsistencies, but do not indicate program defects. However, heuristic techniques can

filter out many of the common cases.

- It can work with or without specifications. Unlike many systems focused on verification, it can work smoothly with assertions that may or may not completely describe the behavior and requirements of the program.

This dissertation has attempted to generalize and formalize existing approaches to inconsistency analysis by building on the foundation of axiomatic semantics, resulting in a technique we call *universal reachability analysis.* This approach to inconsistency detection has a number of desirable properties that go beyond those of inconsistency analysis in general:

- The definition of the analysis is strongly connected to the semantics of the underlying language. Therefore, we can prove that the analysis correctly captures the details of the program being analyzed.

- The same implementation can be used for both inconsistency detection and deductive verification. Under-specified code can be checked for obvious mistakes, and fully-specified code can be verified, simply by varying the parameters to the analysis.

- Universal reachability has a correspondence with the popular testing approach of Condition/Decision coverage. (However, it does not go as far as the more stringent Modified Condition/Decision Coverage (MCDC) standard used for critical aircraft software).

We have built a prototype implementation of universal reachability analysis for C programs, Curate, and evaluated it on a variety of benchmarks, including both robust, widely-used software and synthetic programs known to contain specific bugs. To compare our approach to earlier techniques, we also ran two other tools on the same set of benchmarks. The first, the null pointer analysis included with Saturn, is an earlier implementation of inconsistency detection that focuses on discovering potential null pointer dereferences [37]. The second, the Clang Analyzer, is a simple but efficient bug finding tool included as part of the LLVM Compiler Infrastructure Project [83].

We found that we were able to detect most of the bugs that the Saturn null pointer analysis could find, along with a variety of other bugs that fell outside the scope of this earlier tool. Saturn's analysis also performs a number of checks that fall outside of the realm of inconsistency analysis, as we define it, and these could be useful complements to our approach, as well. Therefore, we are confident that our work generalizes Saturn's approach to inconsistency detection, and that it can be used to discover interesting bugs.

The Clang Analyzer generated warnings that were largely disjoint from those of either Saturn or Curate. In a few cases, Clang was able detect null pointer dereferences that overlapped with one of the other tools. However, the majority of the other messages were either spurious warnings about null pointer dereferences, or about problems outside of the scope of either Saturn or Curate, such as values being written and never read or variables being used without being defined.

The total number of warnings Curate produced was almost always smaller than the number produced by the Clang Analyzer. Therefore, we expect that the number of warnings produced by universal reachability analysis will be low enough to avoid overwhelming programmers. The burden of investigating warnings could be further reduced by limiting the analysis to code already being modified during maintenance, or to code currently being written during initial development.

Our implementation did report a number of inconsistencies that do not represent program defects, but we can place most of these into a small number of well-defined categories, each of which could be eliminated with sufficient engineering effort. Most of the spurious warnings were due to the use of the C pre-processor, which obscures information about the source of expanded macros and platform-specific definitions. Therefore, we expect that the majority of the warnings produced by a production-quality implementation with an integrated pre-processor would indicate true bugs.

In the course of our work, we also developed a heuristic extension to check boundary conditions that yielded some interesting bugs (especially about array bounds), but also generated large numbers of false positives. This heuristic may be valuable in array-heavy code that can be written in a style that avoids the patterns responsible for many false positives.

Curate was not perform as quickly as the Clang Analyzer's simpler checks, which are intended to be appropriate for real-time feedback within an IDE. However, Curate generally took less time than the Saturn null pointer analysis. For both Curate and Saturn's analysis, modern workstations can analyze projects on the order of half a million lines of code overnight. As computing power increases, universal reachability analysis restricted to recently-edited code may be able to provide almost-real-time feedback.

Several directions for additional work are apparent. Most significantly, our work has not investigated interprocedural propagation of analysis information, beyond explicit procedure specifications provided by the programmer and a pre-processing phase to perform exit function identification. The Saturn inconsistency analysis implementation did perform inter-procedural analysis, and found that approximately one third of the inconsistencies they discovered crossed procedure boundaries.

In addition, our implementation was somewhat imprecise, in part because the mathematical techniques we used to model the semantics of C were fairly simple. Newer techniques, particularly those that allow for more sophisticated reasoning about heap structures, could make our analysis both more powerful and more efficient. Reasoning about bit-vectors and floating-point arithmetic may also reveal bugs outside the realm of those we have considered so far.

As a final possible direction for future work, there may be other slicing techniques, perhaps analogous to the boundary condition specialization we explored, that could lead to the discovery of new classes of defects.

Future work in any of these directions would be a valuable addition to the set of available techniques for automated defect detection.

# Appendix A

# Proof of Theorem 1

We adapt contexts $C$ to describe any sub-statement relationship, as well as duplicated sub-statements in sequence. Therefore, contexts $C$ are multi-hole contexts. Only sequences can contain multiple holes, so an alternative exists for $C \ ; \ C$, but not $C \ ! \ C$ or $C \ \square \ C$. Applying a multi-hole context to a statement replaces all holes with that statement.

$$C ::= \bullet \mid C \ ; \ s \mid s \ ; \ C \mid C \ ; \ C \mid C \ ! \ s \mid s \ ! \ C \mid C \ \square \ s \mid s \ \square \ C \mid C^*$$

Slice contexts $D$ can describe any context extension used by the *slices* function. Slice contexts have only a single hole.

$$D ::= \bullet \mid D \ ; \ s \mid (s \ ! \ \bot) \ ; \ D \mid D \ ! \ s \mid (s \ ; \ \bot) \ ! \ D$$

Evaluation context describe contexts in which at least one hole can be evaluated:

$$E ::= \bullet \mid E \ ; \ s \mid E \ ! \ s \mid E \ ; \ C$$

We define evaluation on contexts in Fig. A.1. Contexts are closed under evaluation. Any evaluation of a context results in a context: a statement with at least one hole. Contexts of type $D$ and $E$ are subsets of $C$ and can be evaluated according to the same rules.

For the proof of Theorem 1, we generalize the definition of universal reachability to work within a $D$ context. In this definition, and all of the following proofs, we use the metavariable $r$ to refer to a statement that is either `skip` or `raise`.

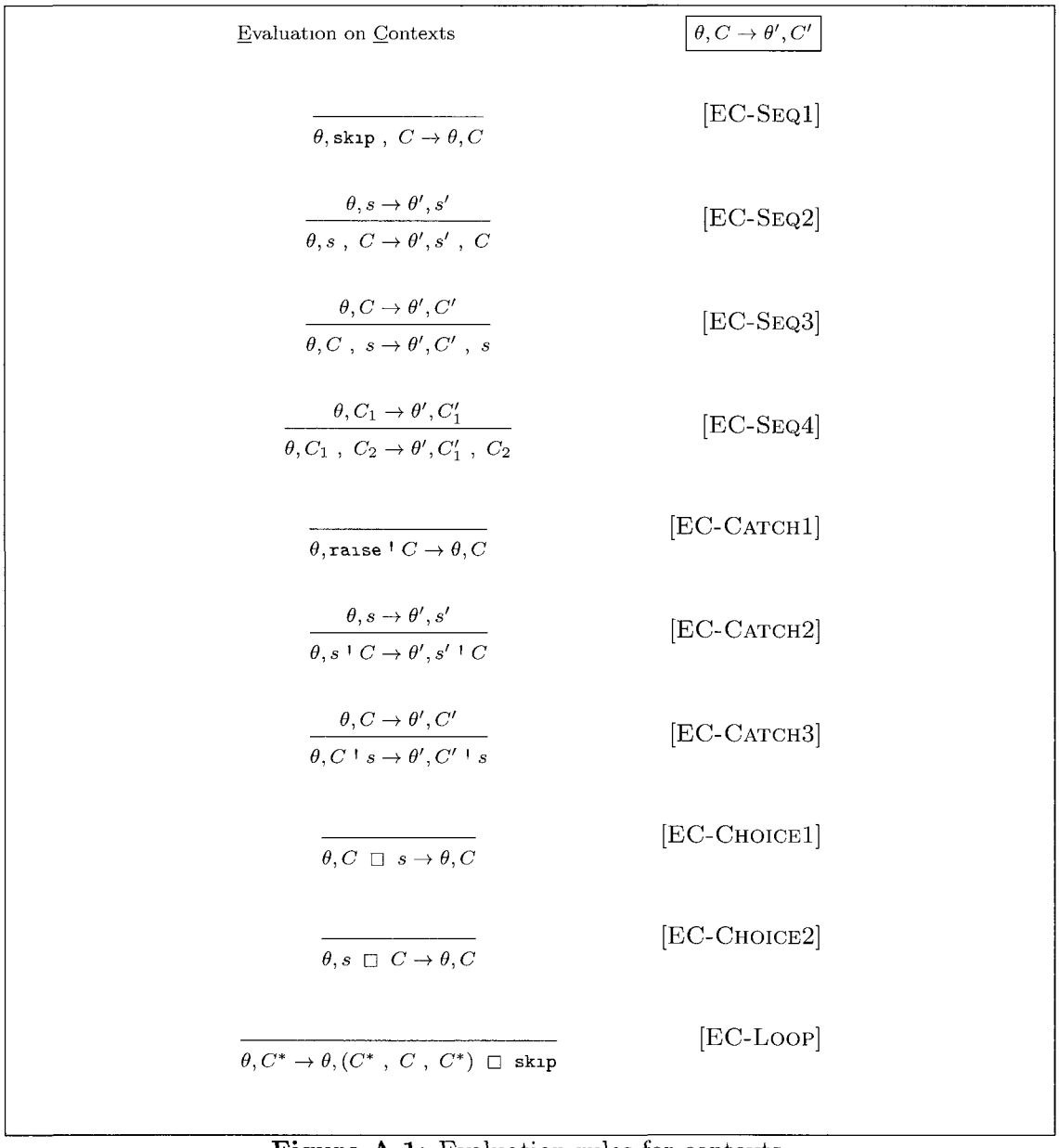


**Figure A.1:** Evaluation rules for contexts.

**Definition 8** (Contextually Universally Reachable). *Given a context $D$ and a statement $s$, we write $UR(D, s)$ if for all $C$ and $t$ such that $s = C[t]$ there exists $E$, $\theta$, $\theta'$, $\theta''$ and $r$ such that $\theta, D[C] \rightarrow^* \theta', E$ and $\theta', E[t] \rightarrow^* \theta'', r$*

If $D = \bullet$, this implies Def. 3 from Sec. 3.1.

**Definition 9.** *If $s$ is terminable, we write $T(s)$ as an abbreviation.*

Next we provide some lemmas about the evaluation of contexts.

**Lemma 1.** *If $\theta, C \ ; \ s \rightarrow^* \theta', E$ then there exists an $E'$ such that $E = E' \ ; \ s$ and $\theta, C \rightarrow^* \theta', E'$*

*Proof.* By induction on the length of the evaluation sequence for $\theta, C \ ; \ s \rightarrow^* \theta', E$

- Case 0. We have that $C \ ; \ s = E$, so $E = E' \ ; \ s$, and $\theta, C \rightarrow^0 \theta', E'$
- Case $n + 1$. Consider the first step of the evaluation sequence: $\theta, C \ ; \ s \rightarrow \theta'', C''$. It can only be via [EC-SEQ3]. Therefore $\theta, C \rightarrow \theta'', C'$, and $\theta'', C' \ ; \ s \rightarrow^* \theta', E$. The final result then holds by induction.

□

**Lemma 2.** *If $\theta, C \ ; \ C' \rightarrow^* \theta', E$ then there exists an $E'$ such that $E = E' \ ; \ C'$ and $\theta, C \rightarrow^* \theta', E'$*

*Proof.* By induction on the length of the evaluation sequence for $\theta, C \ ; \ C' \rightarrow^* \theta', E$

- Case 0. We have that $C \ ; \ C' = E$, so $E = E' \ ; \ C'$, and $\theta, C \rightarrow^0 \theta', E'$
- Case $n+1$. Consider the first step of the evaluation sequence: $\theta, C \ ; \ C' \rightarrow \theta'', C''$. It can only be via [EC-SEQ3]. Therefore $\theta, C \rightarrow \theta'', C'''$, and $\theta'', C''' \ ; \ C' \rightarrow^* \theta', E$. The final result then holds by induction.

□

**Lemma 3.** *If $\theta, s \ ; \ C \rightarrow^* \theta', E$ then there exists a store $\theta''$ such that $\theta, s \rightarrow^* \theta'', \texttt{skip}$ and $\theta, s \ ; \ C \rightarrow^* \theta'', C \rightarrow^* \theta', E$*

*Proof.* By induction on the length of the evaluation sequence $\theta, s\ ;\ C \to^* \theta', E$.

- Case 0. Impossible. The context $s\ ;\ C$ is not an evaluation context.
- Case $n+1$. Case analysis on the first evaluation rule.
  - Case [EC-SEQ1]. Then $s = \texttt{skip}$, $\theta'' = \theta$, and $\theta'', C \to^* \theta', E$.
  - Case [EC-SEQ2]. Then $\theta, s \to \theta''', s'''$, and $\theta''', s'''\ ;\ C \to^* \theta', E$. Therefore $\theta''', s''' \to^* \theta'''', \texttt{skip}$, and $\theta'''', s'''\ ;\ C \to^* \theta'', C \to^* \theta', E$ by induction.

□

**Lemma 4.** *If $\theta, C\ !\ s \to^* \theta', E$ then there exists an $E'$ such that $E = E'\ !\ s$ and $\theta, C \to^* \theta', E'$*

*Proof.* Symmetrical with the proof for Lem. 1, but using [EC-CATCH3] in place of [EC-SEQ3]. □

**Lemma 5.** *If $\theta, s\ !\ C \to^* \theta', E$ then there exists a store $\theta''$ such that $\theta, s \to^* \theta'', \texttt{raise}$ and $\theta, s\ !\ C \to^* \theta'', C \to^* \theta', E$*

*Proof.* Symmetrical with the proof for Lem. 3, but using [EC-CATCH1] in place of [EC-SEQ1], [EC-CATCH2] in place of [EC-SEQ2], and `raise` in place of `skip`. □

**Lemma 6.** *If $\theta, E[s\ ;\ C] \to^* \theta', E'$ then there exists a store $\theta''$ such that $\theta, s \to^* \theta'', \texttt{skip}$ and $\theta, E[s\ ;\ C] \to^* \theta'', E[C] \to^* \theta', E'$*

*Proof.* By induction on the structure of $E$.

- Case $E = \bullet$. Follows from Lem. 3.
- Case $E = E''\ ;\ s''$.

| | | |
|---|---|---|
| $E[s\ ;\ C] = E''[s\ ;\ C]\ ;\ s'' = C'\ ;\ s''$ | Case binding | (1) |
| $\theta, E''[s\ ;\ C]\ ;\ s'' \to^* \theta', E'$ | Assumption | (2) |
| $\theta, E''[s\ ;\ C] \to^* \theta', E'''$ where $E' = E'''\ ;\ s''$ | Lem. 1, (2) | (3) |
| $\theta, E''[s\ ;\ C] \to^* \theta'', E''[C] \to^* \theta', E'''$ | Induction, (3) | (4) |
| $\theta, s \to^* \theta'', \texttt{skip}$ | Induction, (3) | result |
| $\theta, E[s\ ;\ C] \to^* \theta'', E[C] \to^* \theta', E'$ | (1) – (4), [EC-SEQ3] | result |

- Case $E = E''\ !\ s''$. Symmetrical to the previous case, but using Lem. 4 instead of Lem. 1.

- Case $E = E'' \ ; \ C''$.

| | | |
|---|---|---|
| $E[s \ ; \ C] = E''[s \ ; \ C] \ ; \ C'' = C' \ ; \ C''$ | Case binding | (1) |
| $\theta, E''[s \ ; \ C] \ ; \ C'' \rightarrow^* \theta', E'$ | Assumption | (2) |
| $\theta, E''[s \ ; \ C] \rightarrow^* \theta', E'''$ where $E' = E''' \ ; \ C''$ | Lem. 2, (2) | (3) |
| $\theta, E''[s \ ; \ C] \rightarrow^* \theta'', E''[C] \rightarrow^* \theta', E'''$ | Induction, (3) | (4) |
| $\theta, s \rightarrow^* \theta'', \texttt{skip}$ | Induction, (3) | result |
| $\theta, E[s \ ; \ C] \rightarrow^* \theta'', E[C] \rightarrow^* \theta', E'$ | (1) - (4), [EC-SEQ4] | result |

□

**Lemma 7.** *If* $\theta, E[s \ ! \ C] \rightarrow^* \theta', E'$ *then there exists a store* $\theta''$ *such that* $\theta, s \rightarrow^* \theta'', \texttt{raise}$ *and* $\theta, E[s \ ! \ C] \rightarrow^* \theta'', E[C] \rightarrow^* \theta', E'$

*Proof.* Symmetrical with the proof for Lem. 6 but using Lem. 5 in the base case, rather than Lem. 3. □

**Lemma 8.** *If* $\theta, s \ ! \ \bot \rightarrow^* \theta', r$ *then* $\theta, s \rightarrow^* \theta', \texttt{skip}$.

*Proof.* By induction on the length of the derivation of $\theta, s \ ! \ \bot \rightarrow^* \theta', r$.

- Case 0. Impossible, since $s \ ! \ \bot \neq r$.
- Case $n + 1$. We have that $\theta, s \ ! \ \bot \rightarrow \theta'', s' \rightarrow^* \theta', r$. We proceed by case analysis on the first evaluation rule.
  - Case [E-CATCH1]. Then $s = \texttt{raise}$ and $\theta'' = \theta' = \theta$ and $s' = \bot$. But $\theta'', \bot \rightarrow^* \theta', r$ is impossible, so this case cannot occur.
  - Case [E-CATCH2]. Then $s = \texttt{skip}$ and $\theta'' = \theta' = \theta$ and $s' = \texttt{skip}$. Therefore $\theta, s \rightarrow^* \theta', \texttt{skip}$.
  - Case [E-CATCH3]. Then $\theta, s \rightarrow \theta'', s''$ and $s' = s'' \ ! \ \bot$. Therefore the result holds by induction.

□

**Lemma 9.** *If* $\theta, s \ ; \ \bot \rightarrow^* \theta', r$ *then* $\theta, s \rightarrow^* \theta', \texttt{raise}$.

*Proof.* Symmetrical to the proof of Lem. 8. □

**Lemma 10.** *If* $\theta, D[C] \rightarrow^* \theta'', E$ *then there exists* $E'$ *and* $\theta'$ *such that* $\theta, D \rightarrow^* \theta', E'$ *and* $\theta', E'[C] \rightarrow^* \theta'', E$

*Proof.* By induction on D.

- Case $D = \bullet$, choose $E' = \bullet$ and $\theta'' = \theta' = \theta$.

- Case $D = D' \ ; \ s'$

| | | |
|---|---|---|
| $\theta, D'[C] \ ; \ s' \to^* \theta'', E$ | Assumption | (1) |
| $\theta, D'[C] \to^* \theta'', E''$ | By Lem. 1, (1) | (2) |
| $E = E'' \ ; \ s'$ | By Lem. 1, (1) | (3) |
| $\theta, D' \to^* \theta', E'$ | Induction, (2) | (4) |
| $\theta', E'[C] \to^* \theta'', E''$ | Induction, (2) | (5) |
| $\theta, D' \ ; \ s' \to^* \theta', E' \ ; \ s'$ | [EC-SEQ3], transitively, (4) | result |
| $\theta', E'[C] \ ; \ s' \to^* \theta'', E'' \ ; \ s'$ | [EC-SEQ3], transitively, (5) | result |

- Case $D = (s' \ ! \ \bot) \ ; \ D'$

| | | |
|---|---|---|
| $\theta, (s' \ ! \ \bot) \ ; \ D'[C] \to^* \theta'', E$ | Assumption | (1) |
| $\theta, (s' \ ! \ \bot) \ ; \ D'[C] \to^* \theta', D'[C] \to^* \theta'', E$ | By Lem. 3, (1) | (2) |
| $\theta, s' \ ! \ \bot \to^* \theta', \texttt{skip}$ | By Lem. 3, (1) | (3) |
| $\theta, D' \to^* \theta''', E'$ | Induction, (2) | (4) |
| $\theta''', E'[C] \to^* \theta'', E$ | Induction, (2) | (5) |
| $\theta, (s' \ ! \ \bot) \ ; \ D' \to^* \theta', \texttt{skip} \ ; \ D'$ | (3), ev. rules | (6) |
| $\theta', \texttt{skip} \ ; \ D' \to^* \theta', D' \to^* \theta''', E'$ | (4), ev. rules | (7) |
| $\theta, (s' \ ! \ \bot) \ ; \ D' \to^* \theta''', E'$ | (6), (7) | result |
| $\theta''', E'[C] \to^* \theta'', E$ | (5) | result |

- Case $D = D' \ ! \ s'$. Symmetrical with $D = D' \ ; \ s'$, but using Lem. 4 instead of Lem. 1.

- Case $D = (s' \ ; \ \bot) \ ! \ D'$. Symmetrical with $D = (s' \ ! \ \bot) \ ; \ D'$, but using Lem. 5 instead of Lem. 3.

□

**Lemma 11.** *If* $\theta, s \ ; \ s' \to^* \theta', r$ *then there exists an* $\theta''$ *and* $r'$ *such that* $\theta, s \to^* \theta'', r'$ *and* $\theta'', r' \ ; \ s' \to^* \theta', r$.

*Proof.* By induction on the length of the derivation of $\theta, s \ ; \ s' \to^* \theta', s''$.

- Case 0. Impossible because $s \ ; \ s' \neq r$.

- Case $n+1$. We know that $\theta, s \ ; \ s' \to \theta''', s''' \to^* \theta', r$. We proceed by case analysis on the evaluation rule in the first step.

- Case [E-SEQ1]. Then $s = \texttt{skip}$ and $s''' = s'$ and $\theta, s' \rightarrow^* \theta', r$.
- Case [E-SEQ2]. Then $s = \texttt{raise}$ and $s''' = \texttt{raise}$ which can evaluate in zero steps to `raise`.
- Case [E-SEQ3]. Then $\theta, s \rightarrow \theta''', s''''$ and $s''' = s'''' \ ; \ s'$. Since $\theta''', s''' \rightarrow^* \theta', r$, the result holds by induction.

□

**Lemma 12.** *If $\theta, s \ ! \ s' \rightarrow^* \theta', r$ then there exists an $\theta''$ and $r'$ such that $\theta, s \rightarrow^* \theta'', r'$ and $\theta'', r' \ ! \ s' \rightarrow^* \theta', r$.*

*Proof.* Symmetrical with the proof for Lem. 11. □

**Lemma 13.** *If $\theta, (s \ ! \ \bot) \ ; \ s' \rightarrow^* \theta', r$ then there exists an $\theta''$ such that $\theta, s \rightarrow^* \theta'', \texttt{skip}$ and $\theta'', s', \rightarrow^* \theta', r$.*

*Proof.* By induction on the length of the derivation of $\theta, (s \ ! \ \bot) \ ; \ s' \rightarrow^* \theta', r$.

- Case 0. Impossible because $(s \ ! \ \bot) \ ; \ s' \neq r$
- Case $n + 1$. We know that $\theta, (s \ ! \ \bot) \ ; \ s' \rightarrow \theta''', s''' \rightarrow^* \theta', r$. We proceed by case analysis on the evaluation rule in the first step. Because $s \ ! \ \bot$ is not `skip` or `raise`, the first evaluation rule must be [E-SEQ3], so $\theta, s \ ! \ \bot \rightarrow \theta''', s''''$ where $s''' = s'''' \ ; \ s'$. We proceed by case analysis on the evaluation rule used for this step.
  - Case [E-CATCH1]. Then $s = \texttt{raise}$ and $s''' = \bot$. We know that $\bot$ can't evaluate, but we assume $\theta''', s''' \rightarrow^* \theta', r$, so this case is impossible.
  - Case [E-CATCH2]. Then $s = \texttt{skip}$ and $s''' = \texttt{skip}$, which can evaluate in zero steps to `skip`.
  - Case [E-CATCH3]. Then $\theta, s \rightarrow \theta''', s''''$ and $s''' = (s'''' \ ! \ \bot) \ ; \ s'$. Since $\theta''', s''' \rightarrow^* \theta', r$, the result holds by induction.

□

**Lemma 14.** *If $\theta, E[s \ ! \ \bot] \rightarrow^* \theta', r$ then there exists an $\theta''$ such that $\theta, s \rightarrow^* \theta'', \texttt{skip}$ and $\theta'', E[\texttt{skip}], \rightarrow^* \theta', r$.*

*Proof.* By induction on the structure of $E$.

- Case $E = \bullet$. Follows from Lem. 8.

- Case $E = E' \ ; \ s'$.

| | | |
|---|---|---|
| $\theta, E'[s \ ! \ \bot] \ ; \ s' \rightarrow^* \theta', r$ | Assumption | (1) |
| $\theta, E'[s \ ! \ \bot] \rightarrow^* \theta''', r'$ | Lem. 11, (1) | (2) |
| $\theta''', r' \ ; \ s' \rightarrow^* \theta', r$ | Lem. 11, (1) | (3) |
| $\theta, E'[\texttt{skip}] \rightarrow^* \theta''', r'$ | Induction, (2) | (4) |
| $\theta, s \rightarrow^* \theta'', \texttt{skip}$ | Induction, (2) | result |
| $\theta, E'[\texttt{skip}] \ ; \ s' \rightarrow^* \theta', r$ | (3), (4) | result |

- Case $E = E' \ ; \ C'$. Has a hole, so cannot evaluate to $r$.

- Case $E = E' \ ! \ s'$. Symmetrical with the case for $E = E' \ ! \ s'$ but using Lem. 12 instead of Lem. 11.

□

**Lemma 15.** *If $s \in \{i, r\}$ and $\theta, D[s] \rightarrow^* \theta'', r'$ then there exists $E$ and $\theta'$ such that $\theta, D[s] \rightarrow^* \theta', E[s] \rightarrow^* \theta'', r'$*

*Proof.* By induction on the structure of $D$.

- Case $D = \bullet$. Then $E = \bullet$ and the result follows.

- Case $D = D' \ ; \ s''$

| | | |
|---|---|---|
| $\theta, D'[s] \rightarrow^* \theta''', r$ | Lem. 11 | (1) |
| $\theta''', r \ ; \ s'' \rightarrow^* \theta'', r'$ | Lem. 11 | (2) |
| $\theta, D'[s] \rightarrow^* \theta', E'[s]$ | Induction, (1) | (3) |
| $\theta', E'[s] \rightarrow^* \theta''', r$ | Induction, (1) | (4) |
| $\theta, D'[s] \ ; \ s'' \rightarrow^* \theta', E'[s] \ ; \ s''$ | (3), ev. rules | (5) |
| $\theta', E'[s] \ ; \ s'' \rightarrow^* \theta''', r \ ; \ s''$ | (4), ev. rules | (6) |
| $\theta, D'[s] \ ; \ s'' \rightarrow^* \theta', E'[s] \ ; \ s'' \rightarrow^* \theta'', r'$ | (5), (6), (2) | result |

- Case $D = (s'' \ ! \ \bot) \ ; \ D'$

| | | |
|---|---|---|
| $\theta, s'' \rightarrow^* \theta''', \texttt{skip}$ | Lem. 13 | (1) |
| $\theta''', D'[s] \rightarrow^* \theta'', r'$ | Lem. 13 | (2) |
| $\theta''', D'[s] \rightarrow^* \theta', E'[s]$ | Induction, (2) | (3) |
| $\theta', E'[s] \rightarrow^* \theta'', r'$ | Induction, (2) | (4) |
| $\theta, s'' \ ; \ D'[s] \rightarrow^* \theta', E'[s] \rightarrow^* \theta'', r'$ | (1), (3), (4) | result |

- Case $D = D' \ ! \ s''$. Symmetrical with the case for $D = D' \ ; \ s'$.

- Case $D = (s'' \ ; \ \bot) \ ! \ D'$. Symmetrical with the case for $D = (s' \ ! \ \bot) \ ; \ D'$.

□

**Lemma 16.** *If* $\theta, D \rightarrow^* \theta', E$ *then* $\theta, D[C] \rightarrow^* \theta', E[C]$

*Proof.* By induction on the structure of $D$.

- Case $D = \bullet$, $E$ can only equal $\bullet$, $\theta = \theta'$, and $\theta, C \rightarrow^* \theta, C$.

- Case $D = D' \ ; \ s$

| | | |
|---|---|---|
| $D = D' \ ; \ s$ | Assumption | (1) |
| $\theta, D' \ ; \ s \rightarrow^* \theta', E$ | Assumption | (2) |
| $\theta, D' \rightarrow^* \theta', E'$ | By Lem. 1, (2) | (3) |
| $E = E' \ ; \ s$ | By Lem. 1, (2) | (4) |
| $\theta, D'[C] \rightarrow^* \theta', E'[C]$ | Induction, (3) | (5) |
| $\theta, D'[C] \ ; \ s \rightarrow^* \theta', E'[C] \ ; \ s$ | (5), ev. rules | (6) |
| $\theta, D[C] \rightarrow^* \theta', E[C]$ | (1), (4), (6) | result |

- Case $D = (s \ ! \ \bot) \ ; \ D'$

| | | |
|---|---|---|
| $D = (s \ ! \ \bot) \ ; \ D'$ | Assumption | (1) |
| $\theta, (s \ ! \ \bot) \ ; \ D' \rightarrow^* \theta', E$ | Assumption | (2) |
| $\theta, (s \ ! \ \bot) \ ; \ D' \rightarrow^* \theta'', D'$ | By Lem. 3, (2) | (3) |
| $\theta'', D' \rightarrow^* \theta', E$ | By Lem. 3, (2) | (4) |
| $\theta, s \ ! \ \bot \rightarrow^* \theta'', \texttt{skip}$ | By Lem. 3, (2) | (5) |
| $\theta, (s \ ! \ \bot) \ ; \ D'[C] \rightarrow^* \theta'', \texttt{skip} \ ; \ D'[C]$ | (3), ev. rules | (6) |
| $\theta'', \texttt{skip} \ ; \ D'[C] \rightarrow^* \theta'', D'[C]$ | ev. rules | (7) |
| $\theta'', D'[C] \rightarrow^* \theta', E[C]$ | Induction, (4) | (8) |
| $\theta, D[C] \rightarrow^* \theta', E[C]$ | (1), (6), (7), (8) | result |

- Case $D = D' \ ! \ s$. Symmetrical with $D = D' \ ; \ s'$, but using Lem. 4 instead of Lem. 1.

- Case $D = (s \ ; \ \bot) \ ! \ D'$. Symmetrical with $D = (s' \ ! \ \bot) \ ; \ D'$, but using Lem. 5 instead of Lem. 3.

□

**Definition 10** (Hole Filling Similarity)**.** *We write* $C_1 \succ_t C_2$ *if* $C_1$ *can be converted into* $C_2$ *by replacing some, but not all, holes with* $t$*. It is reflexive and transitive, though not symmetric, and defined according to the rules in Fig. A.2.*

$$\frac{}{C \succ_t C} \quad \text{[CF-Refl]}$$

$$\frac{}{C\ ,\ C' \succ_t C[t]\ ,\ C'} \quad \text{[CF-Seq1]}$$

$$\frac{}{C\ ,\ C' \succ_t C\ ,\ C'[t]} \quad \text{[CF-Seq2]}$$

$$\frac{C \succ_t C'}{C\ ,\ s \succ_t C'\ ,\ s} \quad \text{[CF-Seq3]}$$

$$\frac{C \succ_t C'}{s\ ,\ C \succ_t s\ ,\ C'} \quad \text{[CF-Seq4]}$$

$$\frac{C_1 \succ_t C_1' \qquad C_2 \succ_t C_2'}{C_1\ ,\ C_2 \succ_t C_1'\ ,\ C_2'} \quad \text{[CF-Seq5]}$$

$$\frac{C \succ_t C'}{C \upharpoonright s \succ_t C' \upharpoonright s} \quad \text{[CF-Catch1]}$$

$$\frac{C \succ_t C'}{s \upharpoonright C \succ_t s \upharpoonright C'} \quad \text{[CF-Catch2]}$$

$$\frac{C \succ_t C'}{C \ \square\ s \succ_t C' \ \square\ s} \quad \text{[CF-Choice1]}$$

$$\frac{C \succ_t C'}{s \ \square\ C \succ_t s \ \square\ C'} \quad \text{[CF-Choice2]}$$

$$\frac{C \succ_t C'}{C^* \succ_t C'^*} \quad \text{[CF-Loop]}$$

**Figure A.2:** Defining rules for hole filling similarity.

**Lemma 17.** *If $C \succ_t C'$ then $C[t] = C'[t]$.*

*Proof.* By induction on the structure of $C$.

- Case $C = \bullet$. Then $C' = \bullet$ by [CF-REFL], and $\bullet[t] = \bullet[t]$.
- Case $C = C_1 \ ; \ s_2$. Then $C' = C_1' \ ; \ s_2$ where $C_1 \succ_t C_1'$ by [CF-SEQ3]. By induction, $C_1[t] = C_1'[t]$. Therefore $(C_1 \ ; \ s_2)[t] = (C_1' \ ; \ s_2)[t]$.
- Case $C = s_1 \ ; \ C_2$. Symmetrical with case for $C = C_1 \ ; \ s_2$ but using [CF-SEQ4].
- Case $C = C_1 \ ; \ C_2$. Then one of the following cases holds:
  - $C' = C_1[t] \ ; \ C_2$ by [CF-SEQ1]. We have that $C_2 \succ_t C_2$ by [CF-REFL]. Therefore $C_1 \ ; \ C_2 \succ_t C_1[t] \ ; \ C_2$.
  - $C' = C_1 \ ; \ C_2[t]$ by [CF-SEQ2]. We have that $C_1 \succ_t C_1$ by [CF-REFL]. Therefore $C_1 \ ; \ C_2 \succ_t C_1 \ ; \ C_2[t]$.
  - $C' = C_1' \ ; \ C_2'$ where $C_1 \succ_t C_1'$ and $C_2 \succ_t C_2'$ by [CF-SEQ5]. By induction, $C_1[t] = C_1'[t]$ and $C_2[t] = C_2'[t]$. Therefore $(C_1 \ ; \ C_2)[t] = (C_1' \ ; \ C_2')[t]$.
- Case $C = C_1 \ ! \ s_2$. Symmetrical with case for $C = C_1 \ ; \ s_2$ but using [CF-CATCH1].
- Case $C = s_1 \ ! \ C_2$. Symmetrical with case for $C = s_1 \ ; \ C_2$ but using [CF-CATCH2].
- Case $C = C_1 \ \square \ s_2$. Then $C' = C_1' \ \square \ s_2$ where $C_1 \succ_t C_1'$ by [CF-CHOICE1]. By induction, $C_1[t] = C_1'[t]$. Therefore $(C_1 \ \square \ s_2)[t] = (C_1' \ \square \ s_2)[t]$.
- Case $C = s_1 \ \square \ C_2$. Symmetrical with case for $C = C_1 \ \square \ s_2$ but using [CF-CHOICE2].
- Case $C = C_1^*$. Then $C' = C_1'^*$ by [CF-LOOP]. By induction, $C_1[t] = C_1'[t]$. Therefore $C_1^*[t] = C_1'^*[t]$.

□

**Lemma 18.** *If $C_1 \succ_t C_2$ and $\theta, C_2 \to \theta', C_2'$ then either*

1. *There exist $\theta'$ and $C_1'$ such that $\theta, C_1 \to \theta', C_1'$ where $C_1' \succ_t C_2'$, or*

2. *$C_1$ is an evaluation context.*

*Proof.* By induction on the structure of $C_2$. If $C_1$ is an evaluation context, then result (2) holds. So, in the following we assume $C_1$ is not an evaluation context.

- Case $C_2 = \bullet$. No evaluation possible.

- Case $C_2 = C \ ; \ s$.

| | | |
|---|---|---|
| $C_1 = C_{11} \ ; \ s$ | [CF-SEQ3] | (1) |
| $C_{11} \succ_t C$ | [CF-SEQ3] | (2) |
| $C_{11}$ is not an ev. ctx. | $C_1$ is not an ev. ctx. | (3) |
| $\theta, C \to \theta', C'$ | [EC-SEQ3] | (4) |
| $\theta, C \ ; \ s \to \theta', C' \ ; \ s$ | [EC-SEQ3] | (5) |
| $\theta, C_{11} \to \theta', C'_{11}$ | Induction, (2), (3), (4) | (6) |
| $C'_{11} \succ_t C'$ | Induction, (2), (3), (4) | (7) |
| $\theta, C_{11} \ ; \ s \to \theta', C'_{11} \ ; \ s$ | [EC-SEQ3], (6) | result |
| $C'_{11} \ ; \ s \succ_t C' \ ; \ s$ | [CF-SEQ3], (7) | result |

- Case $C_2 = \texttt{skip} \ ; \ C$.

| | | |
|---|---|---|
| $C_1 = \texttt{skip} \ ; \ C_{12}$ | [CF-SEQ4] | (1) |
| $C_{12} \succ_t C$ | [CF-SEQ4] | (2) |
| $\theta, \texttt{skip} \ ; \ C \to \theta, C$ | [EC-SEQ1] | (3) |
| $\theta, \texttt{skip} \ ; \ C_{12} \to \theta, C_{12}$ | [EC-SEQ1] | result |
| $C_{12} \succ_t C$ | (2) | result |

- Case $C_2 = s \ ; \ C$ where $s \neq \texttt{skip}$.

| | | |
|---|---|---|
| $C_1 = s \ ; \ C_{12}$ | [CF-SEQ4] | (1) |
| $C_{12} \succ_t C$ | [CF-SEQ4] | (2) |
| $\theta, s \to \theta', s'$ | [EC-SEQ2] | (3) |
| $\theta, s \ ; \ C \to \theta', s' \ ; \ C$ | [EC-SEQ2], (3) | (4) |
| $\theta, s \ ; \ C_{12} \to \theta', s' \ ; \ C_{12}$ | [EC-SEQ2], (3) | result |
| $s' \ ; \ C_{12} \succ_t s' \ ; \ C$ | [CF-SEQ4], (2) | result |

- Case $C_2 = C \ ; \ C'$.

| | | |
|---|---|---|
| $C_1 = C_{11} \ ; \ C_{12}$ | Def. of $\succ_t$ | (1) |
| $C_{11}$ not an ev. ctx. | $C_1$ is not an ev. ctx. | (2) |
| $C_{11} \succ_t C$ | Def. of $\succ_t$ | (3) |
| $C_{12} \succ_t C'$ | Def. of $\succ_t$ | (4) |
| $\theta, C \ ; \ C' \to \theta', C'' \ ; \ C'$ | [EC-SEQ4] | (5) |
| $\theta, C \to \theta', C''$ | [EC-SEQ4] | (6) |
| $\theta, C_{11} \to \theta', C'_{11}$ | Induction, (3), (6) | (7) |
| $C'_{11} \succ_t C''$ | Induction, (3), (6) | (8) |
| $\theta, C_{11} \ ; \ C_{12} \to \theta', C'_{11} \ ; \ C_{12}$ | [EC-SEQ4], (7) | result |
| $C'_{11} \ ; \ C_{12} \succ_t C'' \ ; \ C'$ | Def. of $\succ_t$, (8), (4) | result |

- Case $C_2 = C \,!\, s$. Symmetrical with the case for $C_2 = C \,;\, s$, but using [EC-Catch3] instead of [EC-Seq3].

- Case $C_2 = s \,!\, C$. Symmetrical with the case for $C_2 = s \,;\, C$, but using [EC-Catch2] instead of [EC-Seq2].

- Case $C_2 = C \;\Box\; s$

| | | |
|---|---|---|
| $C_1 \succ_t C \;\Box\; s$ | Assumption | (1) |
| $C_1 = C' \;\Box\; s$ and $C' \succ_t C$ | [CF-Choice1], (1) | (2) |
| $C_2' = C$ | [E-Choice1] | (3) |
| $C_1' = C'$ | [E-Choice1], (2) | (4) |
| $C_1' \succ_t C_2'$ | (2), (3), (4) | result |

- Case $C_2 = s \;\Box\; C$. Symmetrical with the case for $s \;\Box\; C$ but using [E-Choice2] instead of [E-Choice1].

- Case $C_2 = C^*$

| | | |
|---|---|---|
| $C_1 \succ_t C^*$ | Assumption | (1) |
| $C_1 = C'^*$ | [CF-Loop], (1) | (2) |
| $C' \succ_t C$ | (1), (2) | (3) |
| $C_2' = C^* \,;\, C \,;\, C^*$ | [EC-Loop] | (4) |
| $C_1' = C'^* \,;\, C' \,;\, C'^*$ | [EC-Loop], (2) | (5) |
| $C'^* \,;\, C' \,;\, C'^* \succ_t C^* \,;\, C \,;\, C^*$ | (3), Def. of $\succ_t$ | (6) |
| $C_1' \succ_t C_2'$ | (4), (5), (6) | result |

□

**Lemma 19.** *If $C_1 \succ_t C_2$ and $\theta, C_2 \to^* \theta', C_2'$ then either*

1. *There exist $\theta'$ and $C_1'$ such that $\theta, C_1 \to^* \theta', C_1'$ where $C_1' \succ_t C_2'$, or*

2. *There exist $\theta''$ and $E$ such that $\theta, C_1 \to^* \theta'', E$ where $E \succ_t C_2''$ and $\theta, C_2 \to^* \theta'', C_2'' \to^* \theta', C_2'$*

*Proof.* We consider two cases.

- $C_1$ is an evaluation context. $\theta, C_1 \to^0 \theta, C_1$, and $\theta, C_2 \to^0 \theta, C_2$, and $C_1 \succ_t C_2$.

- $C_1$ is not an evaluation context. We proceed by induction on the length of the evaluation sequence $\theta, C_2 \to^* \theta', C_2'$.

  - Case 0. We have $\theta, C_1 \to^0 \theta, C_1$, and $\theta, C_2 \to^0 \theta, C_2$, and $C_1 \succ_t C_2$.

- Case $n+1$. Consider the first step in the evaluation sequence $\theta, C_2 \to \theta'', C_2'' \to^* \theta', C_2'$. By Lem. 18, we have $\theta, C_1 \to^* \theta'', C_1''$ where $C_1'' \succ_t C_2''$. Therefore, the result for $\theta, C_2'' \to^* \theta', C_2'$ holds by induction.

□

**Lemma 20.** *If $C_1 \succ_t C_2$ and $\theta, C_1 \to \theta', C_1'$ then there exist $\theta'$ and $C_2'$ such that $\theta, C_2 \to \theta', C_2'$ where $C_1' \succ_t C_2'$.*

*Proof.* By induction on the structure of $C_1$.

- Case $C_1 = \bullet$. No evaluation possible.

- Case $C_1 = C \ ; \ s$.

| | | |
|---|---|---|
| $C_2 = C_{21} \ ; \ s$ | [CF-SEQ3] | (1) |
| $C \succ_t C_{21}$ | [CF-SEQ3] | (2) |
| $\theta, C \to \theta', C'$ | [EC-SEQ3] | (3) |
| $\theta, C \ ; \ s \to \theta', C' \ ; \ s$ | [EC-SEQ3] | (4) |
| $\theta, C_{21} \to \theta', C_{21}'$ | Induction, (2), (3) | (5) |
| $C' \succ_t C_{21}'$ | Induction, (2), (3) | (6) |
| $\theta, C_{21} \ ; \ s \to \theta', C_{21}' \ ; \ s$ | [EC-SEQ3], (5) | result |
| $C' \ ; \ s \succ_t C_{21}' \ ; \ s$ | [CF-SEQ3], (6) | result |

- Case $C_1 = s \ ; \ C$.

| | | |
|---|---|---|
| $C_2 = s \ ; \ C'$ | [CF-SEQ4] | (1) |
| $C \succ_t C'$ | [CF-SEQ4] | (2) |
| $\theta, s \to \theta', s'$ | [EC-SEQ2] | (3) |
| $\theta, s \ ; \ C \to \theta', s' \ ; \ C$ | [EC-SEQ2] | (4) |
| $\theta, s \ ; \ C' \to \theta', s' \ ; \ C'$ | [EC-SEQ2], (3) | result |
| $s' \ ; \ C \succ_t s' \ ; \ C'$ | [CF-SEQ4], (2) | result |

- Case $C_1 = C \ ; \ C'$.

  - Case $C_2 = C_{21} \ ; \ C_{22}$.

| | | |
|---|---|---|
| $C \succ_t C_{21}$ | [CF-SEQ5] | (1) |
| $C' \succ_t C_{22}$ | [CF-SEQ5] | (2) |
| $\theta, C \to \theta', C''$ | [EC-SEQ4] | (3) |
| $\theta, C \ ; \ C' \to \theta', C'' \ ; \ C'$ | [EC-SEQ4] | (4) |
| $\theta, C_{21} \to \theta', C_{21}'$ | Induction, (1), (3) | (5) |
| $C'' \succ_t C_{21}'$ | Induction, (1), (3) | (6) |
| $\theta, C_{21} \ ; \ C_{22} \to \theta', C_{21}' \ ; \ C_{22}$ | [EC-SEQ3], (5) | result |
| $C'' \ ; \ C' \succ_t C_{21}' \ ; \ C_{22}$ | (2), (6) | result |

- $C_2 = C[t]\ ;\ C_{22}$.

| | | |
|---|---|---|
| $C' \succ_t C_{22}$ | [CF-SEQ1], [CF-SEQ5] | (1) |
| $\theta, C \to \theta', C''$ | [EC-SEQ4] | (2) |
| $\theta, C\ ;\ C' \to \theta', C''\ ;\ C'$ | [EC-SEQ4] | (3) |
| $\theta, C[t] \to \theta', C''[t]$ | Lem. 22, (2) | (4) |
| $\theta, C[t]\ ;\ C_{22} \to \theta', C''[t]\ ;\ C_{22}$ | [EC-SEQ3], (4) | result |
| $C''\ ;\ C' \succ_t C''[t]\ ;\ C_{22}$ | [CF-SEQ1], [CF-SEQ4], (1) | result |

- $C_2 = C_{21}\ ;\ C'[t]$.

| | | |
|---|---|---|
| $C \succ_t C_{21}$ | [CF-SEQ2], [CF-SEQ5] | (1) |
| $\theta, C \to \theta', C''$ | [EC-SEQ4] | (2) |
| $\theta, C\ ;\ C' \to \theta', C''\ ;\ C'$ | [EC-SEQ4] | (3) |
| $\theta, C_{21} \to \theta', C'_{21}$ | Induction, (1), (2) | (4) |
| $C'' \succ_t C'_{21}$ | Induction, (1), (2) | (5) |
| $\theta, C_{21}\ ;\ C'[t] \to \theta', C'_{21}\ ;\ C'[t]$ | [EC-SEQ3], (4) | result |
| $C''\ ;\ C' \succ_t C'_{21}\ ;\ C'[t]$ | [CF-SEQ2], [CF-SEQ3], (5) | result |

- Case $C_1 = C\ !\ s$. Symmetrical with the case for $C_2 = C\ ;\ s$, but using [EC-CATCH3] instead of [EC-SEQ3], and [CF-CATCH1] instead of [CF-SEQ3].

- Case $C_1 = s\ !\ C$. Symmetrical with the case for $C_2 = s\ ;\ C$, but using [EC-CATCH2] instead of [EC-SEQ2], and [CF-CATCH2] instead of [CF-SEQ4].

- Case $C_1 = C\ \square\ s$. Then $C'_1 = C$ by [EC-CHOICE1] and $C_2 = C'\ \square\ s$ where $C \succ_t C'$ by [CF-CHOICE1]. Therefore $C'_2 = C'$ by [EC-CHOICE1].

- Case $C_1 = s\ \square\ C$. Symmetrical with the previous case, but using [EC-CHOICE2] instead of [EC-CHOICE1] and [CF-CHOICE2] instead of [CF-CHOICE1].

- Case $C_1 = C^*$. Then $C_2 = C'^*$ where $C \succ_t C'$ by [CF-LOOP]. Therefore $C'_2 = C'^*\ ;\ C'\ ;\ C'^*$ by [EC-LOOP].

□

**Lemma 21.** *If $C_1 \succ_t C_2$ and $\theta, C_1 \to^* \theta', C'_1$ then there exist $\theta'$ and $C'_2$ such that $\theta, C_2 \to^* \theta', C'_2$ where $C'_1 \succ_t C'_2$.*

*Proof.* By induction on length of evaluation sequence.

- Case 0. Then $\theta' = \theta$, $C'_1 = C_1$ and $C'_2 = C_2$.

- Case $n+1$. Consider the first step of evaluation, $\theta, C_1 \to \theta'', C_1'' \to^* \theta', C_1'$. By Lem. 20, there exists a $C_2''$ such that $\theta, C_2 \to \theta'', C_2''$ where $C_1'' \succ_t C_2''$. Then, by induction, $\theta'', C_2'' \to^* \theta', C_2'$ where $C_1' \succ_t C_2'$.

□

**Lemma 22.** *If* $\theta, C \to \theta', C'$ *then* $\theta, C[s] \to \theta', C'[s]$

*Proof.* By induction on the structure of $C$.

- Case $C = \bullet$. No evaluation rules apply.
- Case $C = C_1 \ ; \ s_2$. Only [EC-Seq3] can apply. Therefore $\theta, C_1 \to \theta', C_1'$ so $\theta, C_1[s] \to \theta', C_1'[s]$ by induction. Finally, $\theta, C_1[s] \ ; \ s_2 \to \theta', C_1'[s] \ ; \ s_2$ by [E-Seq3].
- Case $C = s_1 \ ; \ C_2$. Only [EC-Seq1] or [EC-Seq2] can apply.
  - Case [EC-Seq1]. Then we have that $s_1 = \texttt{skip}$ and $C' = C_2$. Therefore, $\theta, s_1 \ ; \ C_2[s] \to \theta, C_2[s]$ by [E-Seq1].
  - Case [EC-Seq2]. Then we have that $\theta, s_1 \to \theta', s_1'$ and $C' = s_1' \ ; \ C_2$. Therefore $\theta, s_1 \ ; \ C_2[s] \to \theta', s_1' \ ; \ C_2[s]$ by [E-Seq3].
- Case $C = C_1 \ ; \ C_2$. Only [EC-Seq4] can apply. Therefore $\theta, C_1 \to \theta', C_1'$ and $C' = C_1' \ ; \ C_2$. By induction $\theta, C_1[s] \to \theta', C_1'[s]$. Therefore $\theta, C_1[s] \ ; \ C_2[s] \to \theta', C_1'[s] \ ; \ C_2[s]$ by [E-Seq3].
- Case $C = C_1 \ ! \ s_2$. Symmetrical with the case for $C_1 \ ; \ s_2$.
- Case $C = s_1 \ ! \ C_2$. Symmetrical with the case for $s_1 \ ; \ C_2$.
- Case $C = C_1 \ \square \ s_2$. Only [EC-Choice1] can apply. Then $C' = C_1$, and we have that $\theta, C_1[s] \ \square \ s_2 \to \theta, C_1[s]$ by [E-Choice1].
- Case $C = s_1 \ \square \ C_2$. Symmetrical with the case for $C_1 \ \square \ s_2$.
- Case $C = C_1^*$. Only [EC-Loop] can apply. Therefore, $C' = C_1^* \ ; \ C_1 \ ; \ C_1^* \ \square \ \texttt{skip}$. By [E-Loop], we have $\theta, C_1[s]^* \to \theta, (C_1[s]^* \ ; \ C_1[s] \ ; \ C_1[s]^*) \ \square \ \texttt{skip}$

□

**Lemma 23.** *If* $\theta, C \rightarrow^* \theta', C'$ *then* $\theta, C[s] \rightarrow^* \theta', C'[s]$.

*Proof.* By induction on the length of the evaluation sequence $\theta, C \rightarrow^* \theta', C'$.

- Case 0. We have $C = C'$ so $C[s] = C'[s]$ by Lem. 17.
- Case $n+1$. We have that $\theta, C \rightarrow \theta'', C''$ and $\theta'', C'' \rightarrow^* \theta', C'$. Then by Lem. 22 we have that $\theta, C[s] \rightarrow \theta'', C''[s]$. By induction we have that $\theta'', C''[s] \rightarrow^* \theta', C'[s]$. Therefore $\theta, C[s] \rightarrow^* \theta', C'[s]$.

□

We now use these basic lemmas about evaluation to prove equivalences between the universal reachability of a larger statement and substatements of the form produced by slicing.

**Lemma 24.** $UR(D, s_1 \ ; \ s_2) \Rightarrow UR(D[(s_1 \ ! \ \bot) \ ; \ \bullet], s_2)$

*Proof.* Pick any decomposition of $s_2$ into $C[t]$.

| | | |
|---|---|---|
| $UR(D, s_1 \ ; \ s_2)$ | Assumption | (1) |
| $\theta, D[s_1 \ ; \ C] \rightarrow^* \theta', E'$ | Def. of $UR$, (1) | (2) |
| $\theta', E'[t] \rightarrow^* \theta'', r'$ | Def. of $UR$, (1) | (3) |
| $\theta, D \rightarrow^* \theta''', E''$ | By Lem. 10, (2) | (4) |
| $\theta''', E''[s_1 \ ; \ C] \rightarrow^* \theta', E'$ | By Lem. 10, (2) | (5) |
| $\theta''', s_1 \rightarrow^* \theta'''', \texttt{skip}$ | Lem. 6, (5) | (6) |
| $\theta''', s_1 \ ! \ \bot \rightarrow^* \theta'''', \texttt{skip}$ | (6), ev. rules | (7) |
| $\theta'''', E''[C] \rightarrow^* \theta', E'$ | Lem. 6, (5) | (8) |
| $\theta''', E''[(s_1 \ ! \ \bot) \ ; \ C] \rightarrow^* \theta', E'$ | (7), (8) | (9) |
| $UR(D[(s_1 \ ! \ \bot) \ ; \ \bullet], s_2)$ | (4), (9), (3) | result |

□

**Lemma 25.** $UR(D, s_1 \ ; \ s_2) \Rightarrow UR(D[\bullet \ ; \ s_2], s_1)$

*Proof.* Pick any decomposition of $s_1$ into $C[t]$.

| | | |
|---|---|---|
| $UR(D, s_1 \ ; \ s_2)$ | Assumption | (1) |
| $\theta, D[C \ ; \ s_2] \rightarrow^* \theta', E'$ | Def. of $UR$, (1) | result |
| $\theta', E'[t] \rightarrow^* \theta'', r'$ | Def. of $UR$, (1) | result |

□

**Lemma 26.** $UR(D[(s_1 \ ! \ \bot) \ ; \ \bullet], s_2) \wedge UR(D[\bullet \ ; \ s_2], s_1) \Rightarrow UR(D, s_1 \ ; \ s_2)$

*Proof.* Suppose $s_1 \ ; \ s_2 = C[t]$, and we consider three cases:

1. $C = \bullet$ and $t = s_1 \ ; \ s_2$

| | | |
|---|---|---|
| $UR(D[\bullet \ ; \ s_2], s_1)$ | Assumption | (1) |
| $\theta, D[\bullet \ ; \ s_2] \rightarrow^* \theta', E'$ | Def. of $UR$, (1) | (2) |
| $\theta', E'[s_1] \rightarrow^* \theta'', r'$ | Def. of $UR$, (1) | (3) |
| $\theta, D \rightarrow^* \theta''', E''$ | Lem. 10, (2) | (4) |
| $\theta''', E''[\bullet \ ; \ s_2] \rightarrow^* \theta', E'$ | Lem. 10, (2) | (5) |
| $\theta''', E''[s_1 \ ; \ s_2] \rightarrow^* \theta', E'[s_1]$ | Lem. 23, (5) | (6) |
| $\theta, D[C] \rightarrow^* \theta''', E''$ | (4), $C = \bullet$ | result |
| $\theta''', E''[s_1 \ ; \ s_2] \rightarrow^* \theta'', r'$ | (6), (3) | result |

2. $C = s_1 \ ; \ C'$ and $s_2 = C'[t]$

| | | |
|---|---|---|
| $UR(D[(s_1 \ ! \ \bot) \ ; \ \bullet], s_2)$ | Assumption | (1) |
| $\theta, D[(s_1 \ ! \ \bot) \ ; \ C'] \rightarrow^* \theta', E'$ | Def. of $UR$, (1) | (2) |
| $\theta', E'[t] \rightarrow^* \theta'', r'$ | Def. of $UR$, (1) | (3) |
| $\theta, D \rightarrow^* \theta''', E''$ | Lem. 10, (2) | (4) |
| $\theta''', E''[(s_1 \ ! \ \bot) \ ; \ C'] \rightarrow^* \theta', E'$ | Lem. 10, (2) | (5) |
| $\theta''', s_1 \ ! \ \bot \rightarrow^* \theta'''', \texttt{skip}$ | Lem. 6, (5) | (6) |
| $\theta''', s_1 \rightarrow^* \theta'''', \texttt{skip}$ | Lem. 8, (6) | (7) |
| $\theta''', E''[(s_1 \ ! \ \bot) \ ; \ C'] \rightarrow^* \theta'''', E''[C'] \rightarrow^* \theta', E'$ | Lem. 6, (5) | (8) |
| $\theta''', E''[s_1 \ ; \ C'] \rightarrow^* \theta'''', E''[C'] \rightarrow^* \theta', E'$ | (7), (8) | (9) |
| $\theta, D[C] \rightarrow^* \theta', E'$ | Lem. 16, (4), (9) | result |
| $\theta', E'[t] \rightarrow^* \theta'', r'$ | (3) | result |

3. $C = C' \ ; \ s_2$ and $s_1 = C'[t]$

| | | |
|---|---|---|
| $UR(D[\bullet \ ; \ s_2], s_1)$ | Assumption | (1) |
| $\theta, D[C' \ ; \ s_2] \rightarrow^* \theta', E'$ | Def. of $UR$, (1) | result |
| $\theta', E'[t] \rightarrow^* \theta'', r'$ | Def. of $UR$, (1) | result |

□

**Lemma 27.** $UR(D, s_1 \ ! \ s_2) \Rightarrow UR(D[(s_1 \ ; \ \bot) \ ! \ \bullet], s_2)$

*Proof.* Symmetrical with the proof for Lem. 24 but using Lem. 7 instead of Lem. 6 and Lem. 9 instead of Lem. 8. □

**Lemma 28.** $UR(D, s_1 \ ! \ s_2) \Rightarrow UR(D[\bullet \ ! \ s_2], s_1)$

*Proof.* Symmetrical with the proof for Lem. 25. □

**Lemma 29.** $UR(D[(s_1 \ ; \ \bot) \ ! \ \bullet], s_2) \wedge UR(D[\bullet \ ! \ s_2], s_1) \Rightarrow UR(D, s_1 \ ! \ s_2)$

*Proof.* Symmetrical with the proof for Lem. 26 but using Lem. 7 instead of Lem. 6. □

**Lemma 30.** $UR(D, s_1 \ \square \ s_2) \Rightarrow UR(D, s_1)$

*Proof.* Pick any decomposition of $s_1$ into $C[t]$.

| | | |
|---|---|---|
| $s_1 \ \square \ s_2 = C[t] \ \square \ s_2$ | Substitution | (1) |
| $UR(D, s_1 \ \square \ s_2)$ | Assumption | (2) |
| $UR(D, C[t] \ \square \ s_2)$ | Substitution, (1), (2) | (3) |
| $\theta, D[C \ \square \ s_2] \rightarrow^* \theta', E$ | Def. of $UR$, (3) | (4) |
| $\theta', E[t] \rightarrow^* \theta'', r'$ | Def. of $UR$, (3) | (5) |
| $\theta, D \rightarrow^* \theta''', E'$ | Lem. 10, (4) | (6) |
| $\theta''', E'[C \ \square \ s_2] \rightarrow^* \theta', E$ | Lem. 10, (4) | (7) |
| $\theta, D[C] \rightarrow^* \theta''', E'[C]$ | Lem. 16, (6) | (8) |
| $\theta''', E'[C \ \square \ s_2] \rightarrow \theta''', E'[C] \rightarrow^* \theta', E$ | only ev. rule for (7) | (9) |
| $\theta, D[C] \rightarrow^* \theta''', E'[C] \rightarrow^* \theta', E$ | (8), (9) | (10) |
| $\theta, D[C] \rightarrow^* \theta', E$ | (10) | result |
| $\theta', E[t] \rightarrow^* \theta'', r'$ | (5) | result |

□

**Lemma 31.** $UR(D, s_1 \ \square \ s_2) \Rightarrow UR(D, s_2)$

*Proof.* Symmetrical with Lem. 30. □

**Lemma 32.** $UR(D, s_1) \wedge UR(D, s_2) \Rightarrow UR(D, s_1 \ \square \ s_2)$

*Proof.* Pick a decomposition such that $s_1 \ \square \ s_2 = C[t]$. Three cases are possible:

1. $C = \bullet$ and $t = s_1 \ \square \ s_2$.

| | | |
|---|---|---|
| $UR(D, s_1)$ | Assumption | (1) |
| $\theta, D \rightarrow^* \theta', E$ | Def. of $UR$, (1) | (2) |
| $\theta', E[s_1] \rightarrow^* \theta'', r'$ | Def. of $UR$, (1) | (3) |
| $\theta, D[s_1 \ \square \ s_2] \rightarrow^* \theta', E[s_1 \ \square \ s_2]$ | Lem. 23, (2) | (4) |
| $\theta', E[s_1 \ \square \ s_2] \rightarrow^* \theta', E[s_1]$ | [E-CHOICE1] | (5) |
| $\theta, D \rightarrow^* \theta', E$ | (2) | result |
| $\theta', E[s_1 \ \square \ s_2] \rightarrow^* \theta', r'$ | (3), (5) | result |

2. $C = s_1 \ \square \ C'$ and $s_2 = C'[t]$.

| | | |
|---|---|---|
| $UR(D, s_2)$ | Assumption | (1) |
| $\theta, D[C'] \rightarrow^* \theta', E$ | Def. of $UR$, (1) | (2) |
| $\theta', E[t] \rightarrow^* \theta'', r'$ | Def. of $UR$, (1) | (3) |
| $\theta, D \rightarrow^* \theta''', E'$ | By Lem. 10, (2) | (4) |
| $\theta''', E'[C'] \rightarrow^* \theta', E$ | By Lem. 10, (2) | (5) |
| $\theta, D[s_1 \ \square \ C'] \rightarrow^* \theta''', E'[s_1 \ \square \ C']$ | Lem. 16, (4) | (6) |
| $\theta''', E'[s_1 \ \square \ C'] \rightarrow^* \theta''', E[C']$ | [EC-CHOICE2] | (7) |
| $\theta, D[s_1 \ \square \ C'] \rightarrow^* \theta', E$ | (5),(6),(7) | result |
| $\theta', E[t] \rightarrow^* \theta'', r'$ | (3) | result |

3. $C = C' \ \square \ s_2$ and $s_1 = C'[t]$. Symmetrical with the previous case.

□

**Lemma 33.** $UR(D[(s^* \ ! \ \bot) \ ; \ \bullet \ ; \ s^*], s) \Rightarrow UR(D, s^*)$

*Proof.* Pick any decomposition of $s^*$ into $C[t]$.

1. Case $C = \bullet$ and $t = s^*$.

| | | |
|---|---|---|
| $UR(D[(s^* \ ! \ \bot) \ ; \ \bullet \ ; \ s^*], s)$ | Assumption | (1) |
| $\theta, D[(s^* \ ! \ \bot) \ ; \ \bullet \ ; \ s^*] \rightarrow^* \theta', E'$ | Def. of $UR$, (1) | (2) |
| $\theta', E'[s] \rightarrow^* \theta'', r'$ | Def. of $UR$, (1) | (3) |
| $\theta, D \rightarrow^* \theta''', E$ | By Lem. 10, (2) | (4) |
| $\theta''', E[(s^* \ ! \ \bot) \ ; \ \bullet \ ; \ s^*] \rightarrow^* \theta', E'$ | By Lem. 10, (2) | (5) |
| $\theta''', s^* \rightarrow^* \theta''', s^* \ ; \ s \ ; \ s^*$ | [E-Loop], [E-Choice1] | (6) |
| $\theta''', E[(s^* \ ! \ \bot) \ ; \ s \ ; \ s^*] \rightarrow^* \theta', E'[s] \rightarrow^* \theta'', r'$ | (3), (5), Lem. 23 | (7) |
| $\theta''', s^* \rightarrow^* \theta'''', \texttt{skip}$ | Lem. 14, (7) | (8) |
| $\theta'''', E[\texttt{skip} \ ; \ s \ ; \ s^*] \rightarrow^* \theta', E'[s]$ | Lem. 14, (7) | (9) |
| $\theta''', E[s^* \ ; \ s \ ; \ s^*] \rightarrow^* \theta', E'[s] \rightarrow^* \theta'', r'$ | (7) – (9) | (10) |
| $\theta, D \rightarrow^* \theta''', E$ | (4) | result |
| $\theta''', E[s^*] \rightarrow^* \theta'', r'$ | (6), (10) | result |

2. Case $C = (C')^*$ and $s^* = C'[t]^*$.

| | | |
|---|---|---|
| $UR(D[(s^* \ ! \ \bot) \ ; \ \bullet \ ; \ s^*], s)$ | Assumption | (1) |
| $\theta, D[(s^* \ ! \ \bot) \ ; \ C' \ ; \ s^*] \rightarrow^* \theta', E$ | Def. of $UR$, (1) | (2) |
| $\theta, D \rightarrow^* \theta''', E'$ | Lem. 10, (2) | (3) |
| $\theta''', E'[(s^* \ ! \ \bot) \ ; \ C' \ ; \ s^*] \rightarrow^* \theta', E$ | Lem. 10, (2) | (4) |
| $\theta', E[t] \rightarrow^* \theta'', r'$ | Def. of $UR$, (1) | (5) |
| $\theta''', E'[(C'[t]^* \ ! \ \bot) \ ; \ C' \ ; \ C'[t]^*] \rightarrow^* \theta', E$ | Rewriting (4) | (6) |
| $\theta''', C'[t]^* \rightarrow^* \theta'''', \texttt{skip}$ | Lem. 14, (6) | (7) |
| $\theta'''', E[\texttt{skip} \ ; \ C' \ ; \ C'[t]^*] \rightarrow^* \theta', E$ | Lem. 14, (6) | (8) |
| $\theta''', E'[C'[t]^* \ ; \ C' \ ; \ C'[t]^*] \rightarrow^* \theta', E$ | (7), (8) | (9) |
| $E'[(C')^* \ ; \ C' \ ; \ (C')^*] \succ_t E'[(C'[t])^* \ ; \ C' \ ; \ (C'[t])^*]$ | Def. of $\succ_t$ | (10) |

   By Lem. 19, (9), and (10) there are two possible cases.

   - Case 1. $E'[(C')^* \ ; \ C' \ ; \ (C')^*]$ evaluates just like $E'[s^* \ ; \ C' \ ; \ s^*]$.

| | | |
|---|---|---|
| $\theta''', E'[(C')^* \ ; \ C' \ ; \ (C')^*] \rightarrow^* \theta', E''$ | Lem. 19 | (11) |
| $E'' \succ_t E$ | Lem. 19 | (12) |
| $E''[t] = E[t]$ | Lem. 17, (12) | (13) |
| $\theta, D[(C')^* \ ; \ C' \ ; \ (C')^*] \rightarrow^* \theta', E''$ | Lem. 21, (3), (11) | (14) |
| $\theta, D[(C')^*] \rightarrow^* \theta', E''$ | (14) | result |
| $\theta', E''[t] \rightarrow^* \theta'', r'$ | (5), (13) | result |

- Case 2. $E'[(C')^* \ ; \ C' \ ; \ (C')^*]$ gets stuck at an earlier evaluation context.

| | | |
|---|---|---|
| $\theta''', E'[(C')^* \ ; \ C' \ ; \ (C')^*] \rightarrow^* \theta'''', E''$ | Lem. 19 | (15) |
| $\theta''', E[s^* \ ; \ C' \ ; \ s^* \rightarrow^* \theta'''', C'' \rightarrow^* \theta', E$ | Lem. 19 | (16) |
| $E'' \succ_t C''$ | Lem. 19 | (17) |
| $E''[t] = C''[t]$ | Lem. 17, (17) | (18) |
| $\theta'''', E''[t] \rightarrow^* \theta', E[t]$ | Lem. 23, (16), (18) | (19) |
| $\theta''', E'[(C')^* \ ; \ C' \ ; \ (C')^*] \rightarrow^* \theta'''', E''$ | (16) | (20) |
| $\theta, D[(C')^* \ ; \ C' \ ; \ (C')^*] \rightarrow^* \theta'''', E''$ | Lem. 16, (3), (20) | result |
| $\theta'''', E''[t] \rightarrow^* \theta'', r'$ | (5), (19) | result |

□

**Lemma 34.** $UR(D, s^*) \Rightarrow UR(D[(s^* \ ! \ \bot) \ ; \ \bullet \ ; \ s^*], s)$

*Proof.* Pick a decomposition of s into C[t].

| | | |
|---|---|---|
| $UR(D, s^*)$ | Assumption | (1) |
| $\theta, D[C^*] \rightarrow^* \theta', E$ | Def. of $UR$, (1) | (2) |
| $\theta', E[t] \rightarrow^* \theta'', r'$ | Def. of $UR$, (1) | (3) |
| $\theta, D \rightarrow^* \theta''', E'$ | By Lem. 10, (2) | (4) |
| $\theta''', E'[C^*] \rightarrow^* \theta', E$ | By Lem. 10, (2) | (5) |
| $\theta''', E'[C^*] \rightarrow^* \theta''', E'[C^* \ ; \ C \ ; \ C^*]$ | (5), ev. rules | (6) |
| $\theta''', E'[C^* \ ; \ C \ ; \ C^*] \rightarrow^* \theta', E$ | (5), (6), only ev. rules | (7) |
| $E'[C^* \ ; \ C \ ; \ C^*] \succ_t E'[s^* \ ; \ C \ ; \ s^*]$ | Def. of $\succ_t$ | (8) |
| $\theta, D[(s^* \ ! \ \bot) \ ; \ C \ ; \ s^*] \rightarrow^*$ $\theta''', E'[(s^* \ ! \ \bot) \ ; \ C \ ; \ s^*]$ | Lem. 16, (4) | (9) |
| $\theta''', E'[s^* \ ; \ C \ ; \ s^*] \rightarrow^* \theta', E''$ | Lem. 21, (7), (8) | (10) |
| $\theta''', s^* \rightarrow^* \theta'''', \texttt{skip}$ | Lem. 6, (10) | (11) |
| $\vartheta'''', E'[C \ ; \ s^*] \rightarrow^* \theta', E''$ | Lem. 6, (10) | (12) |
| $\theta''', s^* \ ! \ \bot \rightarrow^* \theta'''', \texttt{skip}$ | ev. rules, (11) | (13) |
| $\theta''', E'[(s^* \ ! \ \bot) \ ; \ C \ ; \ s^*] \rightarrow^* \theta', E''$ | (12), (13) | (14) |
| $E \succ_t E''$ | Lem. 21, (7), (8) | (15) |
| $\theta, D[(s^* \ ! \ \bot) \ ; \ C \ ; \ s^*] \rightarrow^* \theta', E''$ | (9), (14) | result |
| $\theta', E[t] = \theta', E''[t] \rightarrow^* \theta'', r'$ | Lem. 17, (3), (15) | result |

□

We prove Theorem 1 by giving lemmas for each direction separately.

**Lemma 35.** *Given s and D, if each $s'$ in $slices(D, s)$ is terminable then $UR(D, s)$*

*Proof.* By induction on the structure of $s$.

- Case $s \in \{\imath, r\}$

| | | |
|---|---|---|
| $\forall s' \in \mathit{slices}(D, s).\ T(s')$ | Assumption | (1) |
| $\mathit{slices}(D, s) = \{D[s]\}$ | Def of *slices* | (2) |
| $\mathit{substatements}(s) = \{s\}$ | Def. of *substatements* | (3) |
| $\theta, D[s] \rightarrow^* \theta', r'$ | Def. of $T$, (1), (2) | (4) |
| $\theta, D[s] \rightarrow^* \theta'', E[s] \rightarrow^* \theta', r'$ | Lem 15, (4) | result |

- Case $s = s_1 \ \square \ s_2$

| | | |
|---|---|---|
| $\forall s' \in \mathit{slices}(D, s_1 \ \square \ s_2).\ T(s')$ | Assumption | (1) |
| $\forall s' \in \mathit{slices}(D, s_1) \cup \mathit{slices}(D, s_2).\ T(s')$ | Def. of *slices*, (1) | (2) |
| $(\forall s_1' \in \mathit{slices}(D, s_1).\ T(s_1')) \Rightarrow UR(D, s_1)$ | Ind. hyp. | (3) |
| $(\forall s_2' \in \mathit{slices}(D, s_2).\ T(s_1')) \Rightarrow UR(D, s_2)$ | Ind. hyp. | (4) |
| $UR(D, s_1) \wedge UR(D, s_2)$ | (2), (3), (4) | (5) |
| $UR(D, s_1 \ \square \ s_2)$ | Lem. 32, (5) | result |

- Case $s = s_1 \ ; \ s_2$

| | | |
|---|---|---|
| $\forall s' \in \mathit{slices}(D, s_1 \ ; \ s_2).\ T(s')$ | Assumption | (1) |
| $\forall s' \in \mathit{slices}(D[\bullet \ ; \ s_2], s_1).\ T(s')$ | Def. of *slices*, (1) | (2) |
| $\forall s' \in \mathit{slices}(D[(s_1 \ ! \ \bot) \ ; \ \bullet], s_2).\ T(s')$ | Def. of *slices*, (1) | (3) |
| $(\forall s_1' \in \mathit{slices}(D[\bullet \ ; \ s_2], s_1).\ T(s_1')) \Rightarrow$ $UR(D[\bullet \ ; \ s_2], s_1)$ | Ind. hyp. | (4) |
| $(\forall s_2' \in \mathit{slices}(D[(s_1 \ ! \ \bot) \ ; \ \bullet], s_2).\ T(s_1')) \Rightarrow$ $UR(D[(s_1 \ ! \ \bot) \ ; \ \bullet], s_2)$ | Ind. hyp. | (5) |
| $UR(D[\bullet \ ; \ s_2], s_1) \wedge UR(D[(s_1 \ ! \ \bot) \ ; \ \bullet], s_2)$ | (2), (3), (4), (5) | (6) |
| $UR(D, s_1 \ ; \ s_2)$ | Lem. 26, (6) | result |

- Case $s = s_1 \ ! \ s_2$. Symmetrical with the case for $s_1 \ ; \ s_2$ but using Lem. 29 instead of Lem. 26.

- Case $s = s_1^*$

| | | |
|---|---|---|
| $\forall s' \in \mathit{slices}(D, s_1^*).\ T(s')$ | Assumption | (1) |
| $\forall s' \in \mathit{slices}(D[(s_1^* \ ! \ \bot) \ ; \ \bullet \ ; \ s_1^*], s_1).\ T(s')$ | Def. of *slices*, (1) | (2) |
| $(\forall s' \in \mathit{slices}(D[(s_1^* \ ! \ \bot) \ ; \ \bullet \ ; \ s_1^*], s_1).\ T(s')) \Rightarrow$ $UR(D[(s_1^* \ ! \ \bot) \ ; \ \bullet \ ; \ s_1^*], s_1)$ | Ind. hyp. | (3) |
| $UR(D[(s_1^* \ ! \ \bot) \ ; \ \bullet \ ; \ s_1^*], s_1)$ | (2), (3) | (4) |
| $UR(D, s_1^*)$ | Lem. 33, (4) | result |

□

**Lemma 36.** *Given $s$ and $D$, if $UR(D, s)$ then each $s'$ in $\mathit{slices}(D, s)$ is terminable.*

*Proof.* By induction on the structure of $s$.

- Case $s \in \{\imath, r\}$

| | | |
|---|---|---|
| $UR(D, s)$ | Assumption | (1) |
| $slices(D, s) = \{D[s]\}$ | Def. of *slices* | (2) |
| $\theta, D \rightarrow^* \theta', E$ | Def. of $UR$, (1) | (3) |
| $\theta', E[s] \rightarrow^* \theta'', r'$ | Def. of $UR$, (1) | (4) |
| $\theta, D[s] \rightarrow^* \theta', E[s]$ | Lem. 22, (3) | (5) |
| $\theta, D[s] \rightarrow^* \theta'', r'$ | (5), (4) | (6) |
| $T(D[s])$ | Def. of $T$, (6) | result |

- Case $s = s_1 \ \square \ s_2$

| | | |
|---|---|---|
| $UR(D, s_1 \ \square \ s_2)$ | Assumption | (1) |
| $UR(D, s_1)$ | Lem. 30, (1) | (2) |
| $UR(D, s_2)$ | Lem. 31, (1) | (3) |
| $\forall s' \in slices(D, s_1).\ T(s')$ | Induction, (2) | (4) |
| $\forall s' \in slices(D, s_2).\ T(s')$ | Induction, (3) | (5) |
| $\forall s' \in slices(D, s_1 \ \square \ s_2).\ T(s')$ | Def. of *slices*, (4), (5) | result |

- Case $s = s_1 \ ; \ s_2$

| | | |
|---|---|---|
| $UR(D, s_1 \ ; \ s_2)$ | Assumption | (1) |
| $UR(D[\bullet \ ; \ s_2], s_1)$ | Lem. 25, (1) | (2) |
| $UR(D[(s_1 \ ! \ \bot) \ ; \ \bullet], s_2)$ | Lem. 24, (1) | (3) |
| $\forall s' \in slices(D[\bullet \ ; \ s_2], s_1).\ T(s')$ | Induction, (2) | (4) |
| $\forall s' \in slices(D[(s_1 \ ! \ \bot) \ ; \ \bullet], s_2).\ T(s')$ | Induction, (3) | (5) |
| $\forall s' \in slices(D, s_1 \ ; \ s_2).\ T(s')$ | Def. of *slices*, (4), (5) | result |

- Case $s = s_1 \ ! \ s_2$. Symmetrical with the case for $s_1 \ ; \ s_2$, but using Lem. 28 instead of Lem. 25 and Lem. 27 instead of Lem. 24.

- Case $s = s_1^*$

| | | |
|---|---|---|
| $UR(D, s_1^*)$ | Assumption | (1) |
| $UR(D[(s^* \ ! \ \bot) \ ; \ \bullet \ ; \ s^*], s_1)$ | Lem. 34, (1) | (2) |
| $\forall s' \in slices(D[(s^* \ ! \ \bot) \ ; \ \bullet \ ; \ s^*], s_1)$ | Induction, (2) | (3) |
| $\forall s' \in slices(D, s_1^*).\ T(s')$ | Def. of *slices*, (3) | result |

□

We can now go on to prove Theorem 1, which we restate here for convenience.

**Theorem 1.** *Given a statement $s$, the following are equivalent:*

1. *$s$ is universally reachable*

2. *$\forall s' \in slices(\bullet, s)$, $s'$ is terminable*

*Proof.* Follows directly from Lem. 35 and Lem. 36. □

# Appendix B

# Proof of Theorem 2

**Lemma 37.** $\mathit{lfp}(\lambda P.\ N \wedge \mathit{wp}(s, P, X, W)) \Rightarrow N$

*Proof.* Follows from the monotonicity of $\mathit{wp}$. □

**Lemma 38.** $\mathit{lfp}(\lambda P.\ N \wedge \mathit{wp}(s, P, X, W)) \Rightarrow \mathit{wp}(s^*\ ;\ s\ ;\ s^*, N, X, W)$

*Proof.* Let $f(N) = \lambda P.\ N \wedge \mathit{wp}(s, P, X, W)$ and $Q = \mathit{lfp}(f(N)) = \mathit{wp}(s^*, N, X, W)$.

| | | | |
|---|---|---|---|
| $\mathit{wp}(s\ ;\ s^*, N, X, W)$ | $=$ | $\mathit{wp}(s, Q, X, W)$ | Def. of $\mathit{wp}$ and $Q$ |
| | $\Leftarrow$ | $N \wedge \mathit{wp}(s, Q, X, W)$ | Logic |
| | $=$ | $f(N)(Q)$ | Def. of $f$ |
| | $=$ | $Q$ | Def. of $Q$ |
| $\mathit{wp}(s^*\ ;\ s\ ;\ s^*, N, X, W)$ | $=$ | $\mathit{lfp}(f(Q))$ | Def. of $\mathit{wp}$ and $Q$ |
| | $\Leftarrow$ | $\mathit{lfp}(f(N))$ | Since $Q \Rightarrow N$ |
| | $=$ | $Q$ | Since $f$ is monotonic |

□

**Lemma 39.** *For any call-free statement $s$, if $\mathit{wp}(s, N, X, W)(\theta)$ and $\theta, s \to \theta', s'$ then $\mathit{wp}(s', N, X, W)(\theta')$.*

*Proof.* By induction on $s$.

- Case $s = \texttt{assert}(P)$

| | |
|---|---|
| $s' = \texttt{skip}$ | [E-Assert] |
| $\theta' = \theta$ | [E-Assert] |
| $P(\theta)$ | [E-Assert] |
| $\mathit{wp}(s, N, X, W) = (P \wedge N) \vee (\neg P \wedge W)$ | Def. of $\mathit{wp}$ |
| $\mathit{wp}(s', N, X, W) = N$ | Def. of $\mathit{wp}$ |
| $((P \wedge N) \vee (\neg P \wedge W))(\theta) \wedge P(\theta) \Rightarrow N(\theta)$ | Logic |

- Case $s = \texttt{assume}(P)$

| | |
|---|---|
| $s' = \texttt{skip}$ | [E-ASSUME] |
| $\theta' = \theta$ | [E-ASSUME] |
| $P(\theta)$ | [E-ASSUME] |
| $wp(s, N, X, W) = P \Rightarrow N$ | Def. of $wp$ |
| $wp(s', N, X, W) = N$ | Def. of $wp$ |
| $P(\theta) \wedge (P \Rightarrow N)(\theta) \Rightarrow N(\theta)$ | Logic |

- Case $s = x := e$

| | |
|---|---|
| $s' = \texttt{skip}$ | [E-ASSIGN] |
| $\theta' = \theta[x := e(\theta)]$ | [E-ASSIGN] |
| $wp(s, N, X, W) = N[x := e]$ | Def. of $wp$ |
| $wp(s', N, X, W) = N$ | Def. of $wp$ |
| $N[x := e](\theta) \Rightarrow N(\theta[x := e(\theta)])$ | Logic |

- Case $s = s_1 \ \square \ s_2$. Can evaluate via either [E-CHOICE1] or [E-CHOICE2].

  - [E-CHOICE1]

| | |
|---|---|
| $s' = s_1$ | [E-CHOICE1] |
| $\theta' = \theta$ | [E-CHOICE1] |
| $wp(s, N, X, W) = wp(s_1, N, X, W) \wedge wp(s_2, N, X, W)$ | Def. of $wp$ |
| $wp(s', N, X, W) = wp(s_1, N, X, W)$ | Def. of $wp$ |
| $wp(s_1, N, X, W) \wedge wp(s_2, N, X, W) \Rightarrow wp(s_1, N, X, W)$ | Logic |

  - [E-CHOICE2]. Symmetrical with the case for [E-CHOICE1].

- Case $s = \texttt{skip}$. No evaluation rules apply.

- Case $s = s_1 \ ; \ s_2$. Can evaluate via [E-SEQ1], [E-SEQ2], or [E-SEQ3]. In the following, let $Q = wp(s_2, N, X, W)$.

  - [E-SEQ1]

| | |
|---|---|
| $s = \texttt{skip} \ ; \ s_2$ | [E-SEQ1] |
| $s' = s_2$ | [E-SEQ1] |
| $\theta' = \theta$ | [E-SEQ1] |
| $wp(s, N, X, W) = wp(\texttt{skip}, Q, X, W) = Q$ | Def. of $wp$ |
| $wp(s', N, X, W) = Q$ | Def. of $wp$ |

  - [E-SEQ2]

| | |
|---|---|
| $s = \texttt{raise} \ ; \ s_2$ | [E-SEQ2] |
| $s' = \texttt{raise}$ | [E-SEQ2] |
| $\theta' = \theta$ | [E-SEQ2] |
| $wp(s, N, X, W) = wp(\texttt{raise}, Q, X, W) = X$ | Def. of $wp$ |
| $wp(s', N, X, W) = X$ | Def. of $wp$ |

– [E-SEQ3]

| | |
|---|---|
| $\theta, s_1 \to \theta', s_1'$ | [E-SEQ3] |
| $s' = s_1' \ ; \ s_2$ | [E-SEQ3] |
| $wp(s, N, X, W) = wp(s_1, Q, X, W)$ | Def. of $wp$ |
| $wp(s', N, X, W) = wp(s_1', Q, X, W)$ | Def. of $wp$ |
| $wp(s_1, Q, X, W)(\theta) \Rightarrow wp(s_1', Q, X, W)(\theta')$ | Induction |

- Case $s = \texttt{raise}$. No evaluation rules apply.

- Case $s = s_1 \ ! \ s_2$. Can evaluate via [E-CATCH1], [E-CATCH2], or [E-CATCH3]. In the following, let $Q = wp(s_2, N, X, W)$.

  – [E-CATCH1]

  | | |
  |---|---|
  | $s = \texttt{raise} \ ! \ s_2$ | [E-CATCH1] |
  | $s' = s_2$ | [E-CATCH1] |
  | $\theta' = \theta$ | [E-CATCH1] |
  | $wp(s, N, X, W) = wp(\texttt{raise}, N, Q, W) = Q$ | Def. of $wp$ |
  | $wp(s', N, X, W) = Q$ | Def. of $wp$ |

  – [E-CATCH2]

  | | |
  |---|---|
  | $s = \texttt{skip} \ ! \ s_2$ | [E-CATCH2] |
  | $s' = \texttt{skip}$ | [E-CATCH2] |
  | $\theta' = \theta$ | [E-CATCH2] |
  | $wp(s, N, X, W) = wp(\texttt{skip}, N, Q, W) = N$ | Def. of $wp$ |
  | $wp(s', N, X, W) = N$ | Def. of $wp$ |

  – [E-CATCH3]

  | | |
  |---|---|
  | $\theta, s_1 \to \theta', s_1'$ | [E-CATCH3] |
  | $s' = s_1' \ ! \ s_2$ | [E-CATCH3] |
  | $wp(s, N, X, W) = wp(s_1, N, Q, W)$ | Def. of $wp$ |
  | $wp(s', N, X, W) = wp(s_1', N, Q, W)$ | Def. of $wp$ |
  | $wp(s_1, N, Q, W)(\theta) \Rightarrow wp(s_1', N, Q, W)(\theta')$ | Induction |

- Case $s = s_1^*$

| | |
|---|---|
| $s' = (s^* \ ; \ s \ ; \ s^*) \ \square \ \texttt{skip}$ | [E-LOOP] |
| $wp(s, N, X, W) = lfp(\lambda P.\ N \wedge wp(s, P, X, W))$ | Def. of $wp$ |
| $wp(s', N, X, W) = wp(s^* \ ; \ s \ ; \ s^*, N, X, W) \wedge N$ | Def. of $wp$ |
| $wp(s^* \ ; \ s \ ; \ s^*, N, X, W) \wedge N$ | |
| $\Leftarrow lfp(\lambda P.\ N \wedge wp(s, P, X, W))$ | Lem. 37 and Lem. 38 |

□

**Theorem 2.** *If $wp(s, N, X, W) = P$ and $s$ contains no function calls and there exist stores $\theta$ and $\theta'$ and a statement $s'$ such that $\theta, s \to^* \theta', s'$, where $s' \in \{\texttt{skip}, \texttt{raise}\}$, and $\theta$ satisfies $P$ then one of the following cases holds:*

1. $s' = \texttt{skip}$ *and* $\theta'$ *satisfies* $N$

2. $s' = \texttt{raise}$ *and* $\theta'$ *satisfies* $X$

*Proof.* By induction on the length of the evaluation relation.

- Case 0. We have $s = s'$ and $\theta = \theta'$. There are two possible cases.
  - If $s = \texttt{skip}$, then $\theta'$ satisfies $wp(\texttt{skip}, N, X, W) = N$.
  - If $s = \texttt{raise}$, then $\theta'$ satisfies $wp(\texttt{raise}, N, X, W) = X$.
- Case $n + 1$. Holds by application of Lemma 39 and induction.

□

# Appendix C

# Proof of Theorem 3

**Theorem 3.** *For any call-free statement $s$, if $wp(s, \texttt{false}, \texttt{false}, \texttt{true})$ is valid then $s$ is not terminable.*

*Proof.* Assume $wp(s, \texttt{false}, \texttt{false}, \texttt{true})$ is valid. Furthermore, assume that there exist stores $\theta$ and $\theta'$ and a statement $s' \in \{\texttt{skip}, \texttt{raise}\}$ such that $\theta, s \rightarrow^* \theta', s'$. By Theorem 2, $\theta'$ must satisfy `false`. Therefore, such a state cannot exist, so $s$ is not terminable. □

# Appendix D

# Proof of Theorem 4

**Theorem 4.** *Given a call-free statement $s$, if there exists an $s'$ such that $s' \in slices(\bullet, s)$ and $wp(s', \texttt{false}, \texttt{false}, \texttt{true})$ is valid then $s$ is not universally reachable*

*Proof.* If there exists an $s'$ in $slices(\bullet, s)$ such that $wp(s', \texttt{false}, \texttt{false}, \texttt{true})$ is valid then, by Theorem 3, that slice is not terminable. For $s$ to be universally reachable, all of its slices must be terminable, by Theorem 1. Therefore $s$ is not universally reachable. □

# Appendix E

# Proof of Theorem 5

**Definition 11** (Support). *Let the* support *of an unstructured weakest precondition derivation* $v_N, v_X, G \vdash uwp(v_S, N, X, W) = P$ *be a subgraph* $G'$ *of* $G$ *containing all edges and vertices from* $G$ *that participate in paths from* $v_S$ *to either* $v_N$ *or* $v_X$.

**Lemma 40.** *If* $v_N, v_X, G \vdash uwp(v_S, N, X, W) = P$, *and the support of this derivation is a subgraph of* $G'$, *then* $v_N, v_X, G' \vdash uwp(v_S, N, X, W) = P$.

*Proof.* By induction on the derivation of $v_N, v_X, G \vdash uwp(v_S, N, X, W) = P$.

- Case [U-Norm]. Does not depend on the $G$ parameter, so it can be anything.
- Case [U-Exn]. Does not depend on the $G$ parameter, so it can be anything.
- Case [U-Instr].

| | | |
|---|---|---|
| Each path from $v_S$ to $v_N$ in $G$ is in $G'$ | Assumption | (1) |
| Each path from $v_S$ to $v_X$ in $G$ is in $G'$ | Assumption | (2) |
| $v_N, v_X, G \vdash uwp(v', N, X, W) = P'$ | [U-Choice] | (3) |
| $(v_s, v') \in G$ | [U-Choice] | (4) |
| Each path from $v'$ to $v_N$ in $G$ is in $G'$ | (1), (4) | (5) |
| Each path from $v'$ to $v_X$ in $G$ is in $G'$ | (2), (4) | (6) |
| $(v_s, v') \in G'$ | (1), (2) | (7) |
| $v_N, v_X, G' \vdash uwp(v', N, X, W) = P'$ | Induction, (5), (6), (3) | (8) |
| $v_N, v_X, G' \vdash uwp(v_S, N, X, W) = P$ | (7), (8) | result |

- Case [U-CHOICE].

| | | |
|---|---|---|
| Each path from $v_S$ to $v_N$ in $G$ is in $G'$ | Assumption | (1) |
| Each path from $v_S$ to $v_X$ in $G$ is in $G'$ | Assumption | (2) |
| $v_N, v_X, G \vdash uwp(v_1, N, X, W) = P_1$ | [U-CHOICE] | (3) |
| $v_N, v_X, G \vdash uwp(v_2, N, X, W) = P_2$ | [U-CHOICE] | (4) |
| $(v_s, v_1) \in G$ | [U-CHOICE] | (5) |
| $(v_s, v_2) \in G$ | [U-CHOICE] | (6) |
| $P = P_1 \wedge P_2$ | [U-CHOICE] | (7) |
| Each path from $v_1$ to $v_N$ in $G$ is in $G'$ | (1), (5) | (8) |
| Each path from $v_1$ to $v_X$ in $G$ is in $G'$ | (2), (5) | (9) |
| Each path from $v_2$ to $v_N$ in $G$ is in $G'$ | (1), (6) | (10) |
| Each path from $v_2$ to $v_X$ in $G$ is in $G'$ | (2), (6) | (11) |
| $(v_s, v_1) \in G'$ | (1), (2) | (12) |
| $(v_s, v_2) \in G'$ | (1), (2) | (13) |
| $v_N, v_X, G' \vdash uwp(v_1, N, X, W) = P_1$ | Induction, (8), (9), (3) | (14) |
| $v_N, v_X, G' \vdash uwp(v_2, N, X, W) = P_2$ | Induction, (10), (11), (4) | (15) |
| $v_N, v_X, G' \vdash uwp(v_S, N, X, W) = P$ | (12), (13), (14), (15), (7) | result |

□

**Lemma 41.** *If* $v_N, v_X \vdash s \triangleright (v_S, G)$ *and* $v_N, v_X, G \vdash uwp(v_S, N, X, W) = P'$, *and* $wp(s, N, X, W) = P$, *then* $P' \Leftrightarrow P$.

*Proof.* By induction on the structure of $s$.

- Case $s = i$

| | | |
|---|---|---|
| $v_N, v_X \vdash i \triangleright (v_S, G)$ | Assumption | (1) |
| $v_N, v_X, G \vdash uwp(v_S, N, X, W) = P'$ | Assumption | (2) |
| $wp(i, N, X, W) = P$ | Assumption | (3) |
| $G = (\{(v_S, i)\}, \{(v_S, v_N)\}$ | [C-INSTR], (1) | (4) |
| $v_N, v_X, G \vdash uwp(v_N, N, X, W) = N$ | [U-NORM] | (5) |
| $P' = wp(i, N, X, W)$ | [U-INSTR], (2) | (6) |
| $P = wp(i, N, X, W)$ | Def. of $wp$, (3) | (7) |
| $P' \Leftrightarrow P$ | (6), (7) | result |

- Case $s = f()$

| | | |
|---|---|---|
| $v_N, v_X \vdash f() \triangleright (v_S, G)$ | Assumption | (1) |
| $v_N, v_X, G \vdash uwp(v_S, N, X, W) = P'$ | Assumption | (2) |
| $wp(f(), N, X, W) = P$ | Assumption | (3) |
| $\texttt{fun}\ f()\ P_f\ \{\ s'\ \}\ N_f\ X_f \in p$ | Def. of $wp$, (3) | (4) |
| $P = wp(\texttt{assert}(P_f);$ $\texttt{havoctargets}(s');$ $\texttt{assume}(N_f)$ $\square$ $\texttt{assume}(X_f)\ ;\ \texttt{raise},$ $N, X, W)$ | Def. of $wp$, (3) | (5) |
| $G = (V, E)$ | [C-Call], (1) | (6) |
| $V = \{(v_S, i_1), (v_2, i_2), (v_3, i_3), (v_4, i_4)\}$ | [C-Call], (1) | (7) |
| $E = \{(v_S, v_2), (v_2, v_3),$ $(v_2, v_4), (v_3, v_N), (v_4, v_X)\}$ | [C-Call], (1) | (8) |
| $i_1 = \texttt{assert}(P_f)$ | [C-Call], (1) | (9) |
| $i_2 = \texttt{havoctargets}(s')$ | [C-Call], (1) | (10) |
| $i_3 = \texttt{assume}(N_f)$ | [C-Call], (1) | (11) |
| $i_4 = \texttt{assume}(X_f)$ | [C-Call], (1) | (12) |
| $P_4 = wp(i_4, X, X, W)$ | [U-Instr], (6), (7), (8), (12) | (13) |
| $P_3 = wp(i_3, N, X, W)$ | [U-Instr], (6), (7), (8), (11) | (14) |
| $P_2 = wp(i_2, P_3 \wedge P_4, X, W)$ | [U-Choice], (6), (7), (8), (10) | (15) |
| $P' = wp(i_1, P_2, X, W)$ | [U-Instr], (6), (7), (8), (9) | (16) |
| $P' \Leftrightarrow P$ | (5), (13), (14), (15), (16) | result |

- Case $s = s_1 \ \square \ s_2$

| | | |
|---|---|---|
| $v_N, v_X \vdash s_1 \ \square \ s_2 \triangleright (v_S, G)$ | Assumption | (1) |
| $v_N, v_X, G \vdash uwp(v_S, N, X, W) = P'$ | Assumption | (2) |
| $wp(s_1 \ \square \ s_2, N, X, W) = P$ | Assumption | (3) |
| $G = (\{(v_S, \texttt{skip})\}, \{(v_S, v_1), (v_S, v_2)\})$ $\uplus\, G_1 \uplus G_2$ | [C-Choice], (1) | (4) |
| $v_N, v_X \vdash s_1 \triangleright (v_1, G_1)$ | [C-Choice], (1) | (5) |
| $v_N, v_X \vdash s_1 \triangleright (v_2, G_2)$ | [C-Choice], (1) | (6) |
| $G_1 \between G_2$ | [C-Choice], (1) | (7) |
| $v_N, v_X, G \vdash uwp(v_1, N, X, W) = N_1$ | [U-Choice], (2) | (8) |
| $v_N, v_X, G \vdash uwp(v_2, N, X, W) = N_2$ | [U-Choice], (2) | (9) |
| $P' = wp(\texttt{skip}, N_1 \wedge N_2, X, W) = N_1 \wedge N_2$ | [U-Choice], (2) | (10) |
| $P = wp(s_1, N, X, W) \wedge wp(s_2, N, X, W)$ | Def. of $wp$, (3) | (11) |
| Support for (8) is in $G_1$ | (7), (4), (5) | (12) |
| $v_N, v_X, G_1 \vdash uwp(v_1, N, X, W) = N_1$ | Lem. 40, (12), (8) | (13) |
| Support for (9) is in $G_2$ | (7), (4), (6) | (14) |
| $v_N, v_X, G_2 \vdash uwp(v_2, N, X, W) = N_2$ | Lem. 40, (14), (9) | (15) |
| $N_1 \Leftrightarrow wp(s_1, N, X, W)$ | Induction, (5), (13) | (16) |
| $N_2 \Leftrightarrow wp(s_2, N, X, W)$ | Induction, (6), (15) | (17) |
| $P' \Leftrightarrow P$ | (10), (11), (16), (17) | result |

- Case $s = \texttt{skip}$

| | | |
|---|---|---|
| $v_N, v_X \vdash \texttt{skip} \triangleright (v_S, G)$ | Assumption | (1) |
| $v_N, v_X, G \vdash uwp(v_S, N, X, W) = P'$ | Assumption | (2) |
| $wp(\texttt{skip}, N, X, W) = P$ | Assumption | (3) |
| $G = (\{(v_S, \texttt{skip})\}, \{(v_S, v_N)\}$ | [C-Instr], (1) | (4) |
| $v_N, v_X, G \vdash uwp(v_N, N, X, W) = N$ | [U-Norm], (4) | (5) |
| $P' = wp(\texttt{skip}, N, X, W) = N$ | [U-Instr], (2) | (6) |
| $P = N$ | Def. of $wp$, (3) | (7) |
| $P' \Leftrightarrow P$ | (6), (7) | result |

- Case $s = s_1 \ ; \ s_2$

| | | |
|---|---|---|
| $v_N, v_X \vdash s_1 \ ; \ s_2 \triangleright (v_S, G)$ | Assumption | (1) |
| $v_N, v_X, G \vdash uwp(v_S, N, X, W) = P'$ | Assumption | (2) |
| $wp(s_1 \ ; \ s_2, N, X, W) = P$ | Assumption | (3) |
| $P = wp(s_1, wp(s_2, N, X, W), X, W)$ | Def. of $wp$, (3) | (4) |
| $v_2, v_X \vdash s_1 \triangleright (v_S, G_1)$ | [C-SEQ] | (5) |
| $v_N, v_X \vdash s_2 \triangleright (v_2, G_2)$ | [C-SEQ] | (6) |
| $G = G_1 \uplus G_2$ | [C-SEQ] | (7) |
| $G_1 \between G_2$ | [C-SEQ] | (8) |
| Let $v_2, v_X, G_1 \vdash uwp(v_S, N_2, X, W) = P_1$ | Binding | (9) |
| Let $v_N, v_X, G_2 \vdash uwp(v_2, N, X, W) = P_2$ | Binding | (10) |
| $wp(s_1, P_2, X, W) \Leftrightarrow P_1$ | Induction, (5), (9) | (11) |
| $wp(s_2, N, X, W) \Leftrightarrow P_2$ | Induction, (6), (10) | (12) |
| Support for (9) and (10) is in $G$ | (7), (8) | (13) |
| $P_1 \Leftrightarrow P \Leftrightarrow P'$ | (13), (4) | result |

- Case $s =$ `raise`. Symmetrical with the case for `skip`.

- Case $s = s_1 \ ! \ s_2$. Symmetrical with the case for $s_1 \ ; \ s_2$.

- Case $s = s_1^*$. Impossible. We assume $s$ contains no loops.

□

**Theorem 5.** *For any loop-free statement $s$, if $s \blacktriangleright (v_S, v_N, v_X, G)$ and $v_N, v_X, G \vdash uwp(v_S, N, X, W) = P$, then $wp(s, N, X, W) \Leftrightarrow P$.*

*Proof.* Follows directly from Lem. 41. □

# Appendix F

# Proof of Theorem 6

We begin with two lemmas, describing the two parts of Theorem 6.

**Lemma 42.** *If* $v_N, v_X \vdash s \triangleright (v, G)$ *and* $s' \in \mathit{slices}(\bullet, s)$ *and* $\mathit{wp}(s', N, X, W) = P$ *and* $s$ *contains no loops or exceptions, then there exists a* $(v', G')$ *in* $\mathit{uslices}(G)$ *such that* $v_N, v_X, G' \vdash \mathit{uwp}(v', N, X, W) = P'$ *and* $P \Leftrightarrow P'$.

*Proof.* By induction on $s$.

- Case $s = \imath$

| | | |
|---|---|---|
| $v_N, v_X \vdash \imath \triangleright (v, G)$ | Assumption | (1) |
| $s' \in \mathit{slices}(\bullet, \imath)$ | Assumption | (2) |
| $\mathit{wp}(s', N, X, W) = P$ | Assumption | (3) |
| $s' = \imath$ | Def. of *slices*, (2) | (4) |
| $G = (\{(v, \imath)\}, \{(v, v_N)\})$ | [C-Instr], (1) | (5) |
| $\mathit{uslices}(G) = \{G\}$ | Def. of *uslices*, (5) | (6) |
| $v_N, v_X, G \vdash \mathit{uwp}(v, N, X, W) = P'$ | Binding | (7) |
| $P \Leftrightarrow P'$ | Lem. 41, (1), (3), (5) – (7) | result |

- Case $s = f()$

| | | |
|---|---|---|
| $v_N, v_X \vdash f() \triangleright (v, G)$ | Assumption | (1) |
| $s' \in \mathit{slices}(\bullet, f())$ | Assumption | (2) |
| $\mathit{wp}(f(), N, X, W) = P$ | Assumption | (3) |
| $s' = f()$ | Def. of *slices*, (2) | (4) |
| $G = (V, E)$ | [C-Call], (1) | (5) |
| $V = \{(v_S, \imath_1), (v_2, \imath_2),$ $(v_3, \imath_3), (v_4, \imath_4)\}$ | [C-Call], (1) | (6) |
| $E = \{(v_S, v_2), (v_2, v_3),$ $(v_2, v_4), (v_3, v_N), (v_4, v_X)\}$ | [C-Call], (1) | (7) |
| $\mathit{uslices}(G) = \{G\}$ | Def. of *uslices*, (5) – (7) | (8) |
| $v_N, v_X, G \vdash \mathit{uwp}(v, N, X, W) = P'$ | Binding | (9) |
| $P \Leftrightarrow P'$ | Lem. 41, (1), (3), (5) – (9) | result |

- Case $s = s_1 \ \square \ s_2$

| | | |
|---|---|---|
| $v_N, v_X \vdash s_1 \ \square \ s_2 \triangleright (v, G)$ | Assumption | (1) |
| $s' \in \mathit{slices}(\bullet, s_1 \ \square \ s_2)$ | Assumption | (2) |
| $\mathit{wp}(s_1 \ \square \ s_2, N, X, W) = P$ | Assumption | (3) |
| $P = \mathit{wp}(s_1, N, X, W) \wedge \mathit{wp}(s_2, N, X, W)$ | Def. of *wp*, (3) | (4) |
| $G = (\{(v, \texttt{skip})\}, \{(v, v_1), (v, v_2)\}) \cup G_1 \cup G_2$ | [C-Choice], (1) | (5) |
| $v_N, v_X \vdash s_1 \triangleright (v_1, G_1)$ | [C-Choice], (1) | (6) |
| $v_N, v_X \vdash s_1 \triangleright (v_2, G_2)$ | [C-Choice], (1) | (7) |
| $s' \in \mathit{slices}(\bullet, s_1) \cup \mathit{slices}(\bullet, s_2)$ | Def. of *slices*, (2) | (8) |

There are two possible cases:

  - If $s' \in \mathit{slices}(\bullet, s_1)$

| | | |
|---|---|---|
| $G'_1 \in \mathit{uslices}(G_1)$ | Induction, (6) | (9) |
| $v_N, v_X, G'_1 \vdash \mathit{uwp}(v_1, N, X, W) = P'_1$ | Induction, (6) | (10) |
| $\mathit{wp}(s', N, X, W) = P_1$ | Binding | (11) |
| $P_1 \Leftrightarrow P'_1$ | Induction, (6) | (12) |
| Choose $G' = G'_1 \uplus (\{(v, \texttt{skip})\}, \{(v, v_1)\})$ | Def. of *uslices*, (5) | (13) |
| $P \Leftrightarrow P_1$ | (4), (5), (9) – (13) | result |

  - If $s' \in \mathit{slices}(\bullet, s_2)$, the argument is symmetrical.

- Case $s = \texttt{skip}$

| | | |
|---|---|---|
| $v_N, v_X \vdash \texttt{skip} \triangleright (v, G)$ | Assumption | (1) |
| $s' \in \mathit{slices}(\bullet, \texttt{skip})$ | Assumption | (2) |
| $\mathit{wp}(\texttt{skip}, N, X, W) = P$ | Assumption | (3) |
| $s' = \texttt{skip}$ | Def. of *slices*, (2) | (4) |
| $G = (\{(v, \texttt{skip})\}, \{(v, v_N)\})$ | [C-Skip], (1) | (5) |
| $\mathit{uslices}(G) = \{G\}$ | Def. of *uslices*, (5) | (6) |
| $v_N, v_X, G \vdash \mathit{uwp}(v, N, X, W) = P'$ | Binding | (7) |
| $P \Leftrightarrow P'$ | Lem. 41, (1), (3), (5) – (7) | result |

- Case $s = s_1 \ ; \ s_2$

| | | |
|---|---|---|
| $v_N . v_X \vdash s_1 \ ; \ s_2 \triangleright (v, G)$ | Assumption | (1) |
| $s' \in slices(\bullet, s_1 \ ; \ s_2)$ | Assumption | (2) |
| $wp(s_1 \ ; \ s_2, N, X, W) = P$ | Assumption | (3) |
| $P = wp(s_1, wp(s_2, N, X, W), X, W)$ | Def. of $wp$, (3) | (4) |
| Let $P_2 = wp(s_2, N, X, W)$ | Binding | (5) |
| $v_2, v_X \vdash s_1 \triangleright (v_1, G_1)$ | [C-SEQ], (1) | (6) |
| $v_N, v_X \vdash s_2 \triangleright (v_2, G_2)$ | [C-SEQ], (1) | (7) |
| $G = G_1 \cup G_2$ | [C-SEQ], (1) | (8) |
| $s' \in slices(\bullet \ ; \ s_2, s_1) \cup slices(s_1 \ ; \ \bullet, s_2)$ | Def. of $slices$, (2) | (9) |

There are two possible cases:

- If $s' \in slices(\bullet \ ; \ s_2, s_1)$ (10)

| | | |
|---|---|---|
| $s' = s'_1 \ ; \ s_2$ | Def. of $slices$, (10) | (11) |
| $s'_1 \in slices(\bullet, s_1)$ | Def. of $slices$, (10) | (12) |
| $wp(s'_1, P_2, X, W) = P_1$ | Binding | (13) |
| $G'_1 \in uslices(G_1)$ | Induction, (6), (13) | (14) |
| $v_2, v_X, G'_1 \vdash uwp(v_1, P_2, X, W) = P'_1$ | Induction, (6), (13) | (15) |
| $P_1 \Leftrightarrow P'_1$ | Induction, (6), (13) | (16) |
| Choose $G' = G'_1 \uplus G_2$ | Def. of $uslices$, (14), (8) | (17) |
| $v_2, v_X, G' \vdash uwp(v', N, X, W) = P'$ | Binding | (18) |
| $P' = P'_1$ | (15). (17), (18) | (19) |
| $wp(s', N, X, W) \Leftrightarrow P'$ | (13). (16), (19) | result |

- If $s' \in slices(s_1 \ ; \ \bullet, s_2)$ (20)

| | | |
|---|---|---|
| $s' = s_1 \ ; \ s'_2$ | Def. of $slices$, (20) | (21) |
| $s'_2 \in slices(\bullet, s_2)$ | Def. of $slices$, (20) | (22) |
| $wp(s'_2, N, X, W) = P_2$ | Binding | (23) |
| $G'_2 \in uslices(G_2)$ | Induction, (7), (23) | (24) |
| $v_N, v_X, G'_2 \vdash uwp(v_2, N, X, W) = P'_2$ | Induction, (7), (23) | (25) |
| $P_2 \Leftrightarrow P'_2$ | Induction, (7), (23) | (26) |
| $G' = G_1 \uplus G'_2$ | Def. of $uslices$, (24), (8) | (27) |
| $v_2, v_X, G' \vdash uwp(v', N, X, W) = P'$ | Binding | (28) |
| $wp(s', N, X, W) \Leftrightarrow P'$ | (23), (26) | result |

- Case $s =$ `raise`. Symmetrical with the case for `skip`.

- Case $s = s_1 \ ! \ s_2$. Symmetrical with the case for $s_1 \ ; \ s_2$.

- Case $s = s_1^*$ Impossible. We assume $s$ contains no loops.

□

**Lemma 43.** *If* $v_N, v_X \vdash s \triangleright (v, G)$ *and* $(v', G')$ *is in* $\mathit{uslices}(G)$ *and* $v_N, v_X, G' \vdash \mathit{uwp}(v', N, X, W) = P$ *and* $s$ *contains no loops or exceptions then there exists an* $s'$ *in* $\mathit{slices}(\bullet, s)$ *such that* $\mathit{wp}(s', N, X, W) \Leftrightarrow P$.

*Proof.* By induction on $s$.

- Case $s = \imath$. Identical argument to this case in Lem. 42.
- Case $s = f()$. Identical argument to this case in Lem. 42.
- Case $s = s_1 \ \square \ s_2$

  | | | |
  |---|---|---|
  | $v_N, v_X \vdash s_1 \ \square \ s_2 \triangleright (v_S, G)$ | Assumption | (1) |
  | $(v', G') \in \mathit{uslices}(G)$ | Assumption | (2) |
  | $v_N, v_X, G \vdash \mathit{uwp}(v, N, X, W) = P$ | Assumption | (3) |
  | $v_N, v_X \vdash s_1 \triangleright (v_1, G_1)$ | [C-Choice], (1) | (4) |
  | $v_N, v_X \vdash s_2 \triangleright (v_2, G_2)$ | [C-Choice], (1) | (5) |
  | $G = (\{(v, \texttt{skip})\}, \{(v, v_1), (v, v_2)\}) \cup G_1 \cup G_2$ | [C-Choice], (1) | (6) |

  There are four possible cases:

  - If the slice edge is $(v, v_1)$ (7)

    | | | |
    |---|---|---|
    | $G' = (\{(v, \texttt{skip})\}, \{(v, v_1)\}) \uplus G_1$ | Def. of *uslices*, (7) | (8) |
    | $v_N, v_X, G_1 \vdash \mathit{uwp}(v_1, N, X, W) = P_1$ | Binding | (9) |
    | $v_N, v_X, G' \vdash \mathit{uwp}(v, N, X, W) = P' = P_1$ | (8), (9) | (10) |
    | Choose $s' = s_1$ | Binding | (11) |
    | $\mathit{wp}(s_1, N, X, W) = P_1 = P'$ | Thm. 5, (4), (9), (11) | result |

  - If the slice edge is $(v, v_2)$, the proof is symmetrical to the case for slice edge $(v, v_1)$.
  - If the slice edge is in $G_1$ (12)

    | | | |
    |---|---|---|
    | $G'_1 \in \mathit{uslices}(G_1)$ | Def. of *uslices*, (12) | (13) |
    | $G' = (\{(v, \texttt{skip})\}, \{(v, v_1)\}) \uplus G'_1$ | Def. of *uslices*, (12) | (14) |
    | $v_N, v_X, G_1 \vdash \mathit{uwp}(v_1, N, X, W) = P'_1$ | Binding | (15) |
    | $\exists s' \in \mathit{slices}(\bullet, s_1)$ $\mathit{wp}(s', N, X, W) \Leftrightarrow P'_1 = P$ | Induction, (4), (15) | result |

  - If the slice edge is in $G_2$, the proof is symmetrical to the case for a slice edge in $G_1$.
- Case $s = \texttt{skip}$. Identical argument to this case in Lem. 42.

- Case $s = s_1 \ ; \ s_2$

| | | |
|---|---|---|
| $v_N, v_X \vdash s_1 \ ; \ s_2 \triangleright (v_S, G)$ | Assumption | (1) |
| $G' \in \mathit{uslices}(G)$ | Assumption | (2) |
| $v_N, v_X, G' \vdash \mathit{uwp}(v, N, X, W) = P$ | Assumption | (3) |
| $v_2, v_X \vdash s_1 \triangleright (v_1, G_1)$ | [C-SEQ], (1) | (4) |
| $v_N, v_X \vdash s_2 \triangleright (v_2, G_2)$ | [C-SEQ], (1) | (5) |
| $G = G_1 \cup G_2$ | [C-SEQ], (1) | (6) |

  There are two possible cases:

  - If the slice edge is in $G_1$ (7)

| | | |
|---|---|---|
| $G' = G'_1 \uplus G_2$ | Def. of $\mathit{uslices}$, (7) | (8) |
| $G'_1 \in \mathit{uslices}(G_1)$ | Def. of $\mathit{uslices}$, (7) | (9) |
| $v_N, v_X, G_2 \vdash \mathit{uwp}(v_2, N, X, W) = P_2$ | Binding | (10) |
| $\mathit{wp}(s_2, N, X, W) \Leftrightarrow P_2$ | Thm. 5, (4), (10) | (11) |
| $v_2, v_X, G'_1 \vdash \mathit{uwp}(v_1, P_2, X, W) = P'_1$ | Binding | (12) |
| $\exists s'_1 \in \mathit{slices}(\bullet, s_1).$ $\mathit{wp}(s'_1, P_2, X, W) = P'_1$ | Induction, (10), (9), (4) | (13) |
| Choose $s' = s'_1 \ ; \ s_2$ | Binding | (14) |
| $P'_1 = \mathit{wp}(s'_1, \mathit{wp}(s_2, N, X, W), X, W)$ | (11), (13), (14) | result |

  - If the slice edge is in $G_2$ (15)

| | | |
|---|---|---|
| $G' = G_1 \uplus G'_2$ | Def. of $\mathit{uslices}$, (15) | (16) |
| $G'_2 \in \mathit{uslices}(G_2)$ | Def. of $\mathit{uslices}$, (15) | (17) |
| $v_N, v_X, G'_2 \vdash \mathit{uwp}(v_2, N, X, W) = P'_2$ | Binding | (18) |
| $v_2, v_X, G_1 \vdash \mathit{uwp}(v_1, P'_2, X, W) = P_1$ | Binding | (19) |
| $\mathit{wp}(s_1, P'_2, X, W) \Leftrightarrow P_1$ | Thm. 5, (5), (19) | (20) |
| $\exists s'_2 \in \mathit{slices}(\bullet, s_2).$ $\mathit{wp}(s'_2, N, X, W) = P'_2$ | Induction, (18), (17), (5) | (21) |
| Choose $s' = s_1 \ ; \ s'_2$ | Binding | (22) |
| $P'_2 = \mathit{wp}(s_1, \mathit{wp}(s'_2, N, X, W), X, W)$ | (20) – (22) | result |

- Case $s = \texttt{raise}$. Identical argument to this case in Lem. 42.

- Case $s = s_1 \ ! \ s_2$. Symmetrical with the case for $s_1 \ ; \ s_2$.

- Case $s = s_1^*$. Identical argument to this case in Lem. 42.

□

**Theorem 6.** *The following two statements are true of any loop-free and exception-free statement $s$:*

- *If $s' \in slices(\bullet, s)$ and $s \blacktriangleright (v_S, v_N, v_X, G)$ then there exists a $G' \in uslices(G)$ such that $v_N, v_X, G' \vdash uwp(v_S, N, X, W) = P$ and $wp(s', N, X, W) \Leftrightarrow P$.*
- *If $s \blacktriangleright (v_S, v_N, v_X, G)$ and $G' \in uslices(G)$ and $v_N, v_X, G' \vdash uwp(v_S, N, X, W) = P$ then there exists an $s' \in slices(\bullet, s)$ such that $wp(s', N, X, W) \Leftrightarrow P$.*

*Proof.* Follows immediately from Lem. 42 and Lem. 43. □

# Bibliography


[1] Frama-C. `http://frama-c.com`.

[2] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, 1991.

[3] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2nd edition, August 2006.

[4] Alexander Aiken. Introduction to set constraint-based program analysis. *Science of Computer Programming*, 35(2-3):79 – 111, 1999.

[5] Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 163–173, 1994.

[6] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, volume 37, pages 4–16, January 2002.

[7] Lennart Augustsson. Cayenne—a language with dependent types. In *ICFP '98: Proceedings of the third ACM SIGPLAN International Conference on Functional Programming*, pages 239–250, New York, NY, USA, 1998. ACM.

[8] J. W. Backus. F. L. Bauer, J. Green, C. Katz, J. McCarthy, P. Naur, A. J. Perlis, H. Rutishauser, K. Samuelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Revised Report on the Algorithmic Language ALGOL 68, September 1973.

[9] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *International SPIN Workshop on Model Checking of Software*, pages 103–122, May 2001.

[10] Mike Barnett, Bor-Yuh E. Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Formal Methods for Components and Objects*, Lecture Notes in Computer Science, chapter 17, pages 364–387. Springer-Verlag, 2006.

[11] Mike Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In *Workshop on Program Analysis for Software Tools and Engineering*, 2005.

[12] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *Proceedings of the International Workshop on the Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, 2004.

[13] Josh Berdine, Cristiano Calcagno, and Peter O'hearn. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. pages 115–137. 2006.

[14] Gilad Bracha. Pluggable type systems, October 2004.

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

[16] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In Richard Draves, Robbert van Renesse, Richard Draves, and Robbert van Renesse, editors. *Proceedings of the International Conference on Operating Systems Design and Implementation*, pages 209–224. USENIX Association, 2008.

[17] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: automatically generating inputs of death. In *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*, pages 322–335, New York, NY, USA, 2006. ACM Press.

[18] Robert Cartwright and Mike Fagan. Soft typing. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 278–292, New York, NY, USA, 1991. ACM.

[19] Alonzo Church. A Set of Postulates for the Foundation of Logic. *Annals of Mathematics*, 2(33):346–366, 1932.

[20] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP '00: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, pages 268–279, New York, NY, USA, 2000. ACM.

[21] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-Guided Abstraction Refinement. In *CAV '00: Proceedings of the 12th International Conference on Computer Aided Verification*, pages 154–169, London, UK, 2000. Springer-Verlag.

[22] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 2000.

[23] Ernie Cohena, Michał Moskal, Wolfram Schulte, and Stephan Tobies. A precise yet efficient memory model for C. In *Proceedings of the International Workshop on Systems Software Verification*, May 2009.

[24] Sylvain Conchon, Evelyne Contejean, and Johannes Kanig. Ergo : a theorem prover for polymorphic first-order logic modulo theories, 2006.

[25] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Abstraction-refinement for termination. In *Proceedings of the Static Analysis Symposium*, pages 87 – 101, 2005.

[26] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 415–426, New York, NY, USA, 2006. ACM.

[27] Coverity. Prevent. `http://www.coverity.com/products/coverity-prevent.html`.

[28] Flaviu Cristian. Exception handling. In *Dependability of Resilient Computers*, pages 68–97, 1989.

[29] Christoph Csallner and Yannis Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software — Practice and Experience*, 34(11):1025–1050, September 2004.

[30] Christoph Csallner and Yannis Smaragdakis. Check 'n' Crash: combining static checking and testing. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, pages 422–431, New York, NY, USA, 2005. ACM Press.

[31] Christoph Csallner, Yannis Smaragdakis, and Tao Xie. DSD-Crasher: A hybrid analysis tool for bug finding. *ACM Transactions on Software Engineering Methodology*, 17(2):1–37, April 2008.

[32] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. DySy: dynamic symbolic execution for invariant inference. In *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, pages 281–290, New York, NY, USA, 2008. ACM.

[33] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[34] David L. Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. Technical report, HP Labs, 2003.

[35] Edsger W. Dijkstra. Letters to the editor: go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, March 1968.

[36] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Inc., October 1976.

[37] Isil Dillig, Thomas Dillig, and Alex Aiken. Static error detection using semantic inconsistency inference. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 435–445, New York, NY, USA, 2007. ACM.

[38] Dawson Engler. Research web site. `http://www.stanford.edu/~engler/`.

[39] Dawson Engler, David Y. Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. *SIGOPS Operating Systems Review*, 35(5):57–72, December 2001.

[40] Dawson Engler and Madanlal Musuvathi. Static Analysis versus Software Model Checking for Bug Finding. In *VMCAI'04: Proceedings of the 5th International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 405–427, 2004.

[41] Michael D. Ernst. *Dynamically Discovering Likely Program Invariants.* Ph.d., University of Washington Department of Computer Science and Engineering, Seattle, Washington, August 2000.

[42] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, 2002.

[43] Jean-Christophe Filliâtre. Why: a multi-language multi-prover verification condition generator. Research Report 1366, LRI, Université Paris Sud, March 2003.

[44] Cormac Flanagan. Hybrid type checking. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, pages 245–256, January 2006.

[45] Cormac Flanagan, Stephen N. Freund, and Aaron Tomb. Hybrid types, invariants, and refinements for imperative objects. In *Workshop on Foundations and Developments of Object-Oriented Languages*, January 2006.

[46] Cormac Flanagan and K. Rustan M. Leino. Houdini, an Annotation Assistant for ESC/Java. In *FME '01: Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, pages 500–517, London, UK, 2001. Springer-Verlag.

[47] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 234–245, June 2002.

[48] Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, pages 193–205, January 2001.

[49] Robert W. Floyd. Assigning meanings to programs. In *Proceedings of the American Mathematical Society Symposia on Applied Mathematics*, volume 19, pages 19–31. American Mathematical Society, Providence, R.I., 1967.

[50] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 213–223, New York, NY, USA, June 2005. ACM Press.

[51] GrammaTech. Codesonar. `http://grammatech.com/products/codesonar/overview.html`.

[52] Radu Grigore, Julien Charles, Fintan Fairmichael, and Joseph Kiniry. Strongest postcondition of unstructured programs. In *FTfJP '09: Proceedings of the 11th International Workshop on Formal Techniques for Java-like Programs*, pages 1–7, New York, NY, USA, 2009. ACM.

[53] Jessica Gronski, Kenneth Knowles, Aaron Tomb, Stephen N. Freund, and Cormac Flanagan. Sage: Practical hybrid checking for expressive types and specifications. Technical report, University of California, Santa Cruz, 2006.

[54] HackageDB. `language-c`. `http://hackage.haskell.org/package/language-c`.

[55] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, George C. Necula, Grégoire Sutre, and Westley Weimer. Temporal-safety proofs for systems code. In *Proceedings of the IEEE Conference on Computer Aided Verification*, pages 526–538, July 2002.

[56] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[57] Gerard J. Holzmann, Rajeev Joshi, and Alex Groce. New challenges in model checking. In *25 Years of Model Checking*, volume 5000 of *Lecture Notes in Computer Science*, pages 65–76, 2008.

[58] David Hovemeyer and William Pugh. Finding bugs is easy. In *Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 132–136, 2004.

[59] RTCA Inc. DO-178B, software considerations in airborne systems and equipment certification.

[60] SRI International. Yices SMT solver. `http://yices.csl.sri.com/`.

[61] Stephen Johnson. Lint, a C program checker. Computer Science Technical Report 65, Bell Laboratories, December 1977.

[62] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. *Communications of the ACM*, 53(6), June 2010.

[63] Klockwork. Insight. `http://www.klocwork.com/products/insight/`.

[64] Claude Marché, Christine Paulin-Mohring, and Xavier Urbain. The Krakatoa tool for certification of Java/JavaCard programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1-2):89–106, 2004.

[65] Simon Marlow and Philip Wadler. A practical subtyping system for Erlang. In *Proceedings of the 2nd International Conference on Functional Programming (ICFP'97)*, June 1997.

[66] Digital Mars. The D programming language. `http://www.digitalmars.com/d/`.

[67] Guillaume Melquiond. Gappa: Génération automatique de preuves de propriétés arithmétiques. `http://gappa.gforge.inria.fr/`.

[68] Bertrand Meyer. *Eiffel: The Language.* Prentice-Hall, 1992.

[69] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

[70] Yannick Moy. Sufficient preconditions for modular assertion checking. In *Proceedings of the International Conference on Verification, Model Checking and Abstract Interpretation*, pages 188–202, 2008.

[71] Aleksandar Nanevski and Greg Morrisett. Dependent type theory of stateful higher-order functions. Technical Report TR-24-05, Harvard University, 2005.

[72] Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. Polymorphism and separation in Hoare Type Theory. In *ICFP '06: Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, pages 62–73, New York, NY, USA, 2006. ACM.

[73] Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. Ynot: dependent types for imperative programs. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN International Conference on Functional Programming*, pages 229–240, New York, NY, USA, 2008. ACM.

[74] George C. Necula, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy code. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, pages 128–139, 2002.

[75] Charles G. Nelson. *Techniques for program verification.* Stanford University, 1980.

[76] Greg Nelson. A generalization of Dijkstra's calculus. *ACM Transactions on Programming Languages and Systems*, 11(4):517–561, October 1989.

[77] Jeremy W. Nimmer and Michael D. Ernst. Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java. In *Proceedings of the Workshop on Runtime Verification*, Paris, France, July 2001.

[78] Tobias Nipkow. Hoare Logics in Isabelle/HOL. In *Proof and System-Reliability*, pages 341–367, 2002.

[79] National Institute of Standards and Technology. SAMATE: Software Assurance Metrics And Tool Evaluation. `http://samate.nist.gov/`.

[80] Peter O'Hearn, John Reynolds, and Hongseok Yang. Local Reasoning about Programs that Alter Data Structures. In Laurent Fribourg, editor, *Computer Science Logic*, volume 2142 of *Lecture Notes in Computer Science*, chapter 1, pages 1–19. Springer Berlin Heidelberg, Berlin, Heidelberg, August 2001.

[81] Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. Dynamic typing with dependent types. In *IFIP International Conference on Theoretical Computer Science*, pages 437–450, August 2004.

[82] The GNU Project. The Sather programming language. `http://www.gnu.org/software/sather`.

[83] The LLVM Project. Clang static analyzer. `http://clang-analyzer.llvm.org/`.

[84] The LogiCal Project. The Coq proof assistant. `http://coq.inria.fr`.

[85] Grigore Roşu, Chucky Ellison, and Wolfram Schulte. Matching logic: An alternative to Hoare/Floyd logic. In *Thirteenth International Conference on Algebraic Methodology And Software Technology (AMAST '10)*, volume 6486. LNCS, 2010.

[86] PLT Scheme. `http://www.plt-scheme.org/`.

[87] Koushik Sen. Concolic testing. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, pages 571–572, 2007.

[88] Jeremy Siek and Walid Taha. Gradual Typing for Objects. In *ECOOP '07: Proceedings of the 21st European conference on ECOOP 2007*, pages 2–27, Berlin, Heidelberg, 2007. Springer-Verlag.

[89] Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, September 2006.

[90] Jeremy G. Siek and Manish Vachharajani. Gradual typing with unification-based inference. In *DLS '08: Proceedings of the 2008 symposium on Dynamic languages*, pages 1–12, New York, NY, USA, 2008. ACM.

[91] Fortify Software. Fortify 360. `http://www.fortify.com/products/fortify-360/`.

[92] Aaron Tomb, Guillaume Brat, and Willem Visser. Variably interprocedural program analysis for runtime error detection. In *Proceedings of the International Symposium on Software Testing and Analysis*, 2007.

[93] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[94] Hongwei Xi. Imperative programming with dependent types. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, pages 375–387, June 2000.

[95] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 214–227, New York, NY, USA, 1999. ACM Press.

[96] Junfeng Yang, Can Sar, Paul Twohey, Cristian Cadar, and Dawson Engler. Automatically generating malicious disks using symbolic execution. In *2006 IEEE Symposium on Security and Privacy*, pages 15+, 2006.