Program Analysis using Random Interpretation

by

Sumit Gulwani

B.Tech. (Indian Institute of Technology, Kanpur) 2000

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

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:
Professor George C. Necula, Chair
Professor Rastislav Bodik
Professor Bjorn Poonen

Fall 2005

Program Analysis using Random Interpretation

Abstract

Program Analysis using Random Interpretation

by

Sumit Gulwani
Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor George C. Necula, Chair

Random interpretation is a new program analysis technique that uses the power of randomization to verify and discover program properties. It is inspired by, and combines the strengths of, the two complementary techniques for program analysis: *random testing* and *abstract interpretation*. Random testing is simple and finds real bugs in programs, but cannot prove absence of bugs. Abstract interpretation, on the other hand, is a class of sound and deterministic program analyses that find all bugs, but also report spurious bugs (false positives). Often these analyses are complicated and have long running time. This thesis describes few random interpretation based program analyses that are more efficient as well as simpler than their deterministic counterparts that had been state-of-the-art for almost 30 years. This thesis also describes how to extend some of these intraprocedural analyses to an interprocedural setting. We also discuss our experience experimenting with these algorithms.

Professor George C. Necula
Dissertation Committee Chair

For ...

Contents

List of Figures

List of Tables

Acknowledgments

Thanks to the chance discussion with Charles Consel (during my summer internship at IRISA, Rennes) wherein he suggested I might enjoy working with "proof-carrying code" George Necula for my graduate studies. George's philosophy of "I agree that the cost of living here is high, but this is because the place is so beautiful that everyone in the world wants to live here!" was cool enough to attract me to Berkeley.

I have been extremely lucky to do 5 years of apprenticeship under George. I could not have found a better adviser, in spite of the fact that George himself was new to this role when I started working with him. George has been very supportive of my adventures into random research territories. He involved himself in teaching me the entire spectrum of skills needed to be a complete researcher, ranging from choosing interesting research problems and formalizing crude ideas to the art of writing research papers, preparing presentation slides and giving presentations. In each weekly meeting, there was much more to be gained from the brilliance and wisdom of George, which extends far beyond computer science. He could teach how to repair cars, how to build a home-office, give tips for parking in busy shopping districts, . . . , anything that one would want to know under the sun. George and Simona have been very kind to consider us part of their family inviting us regularly to their thanksgiving dinners, their culinary adventures with different cuisines, and sailing expeditions. George has been more than an adviser to me. He has always been there like an elder brother to listen to all my problems. He has taught me finer points in life beyond academics. He is my role model. Thanks George for all that you have done for me, and all that you have been to me!

Several people have asked me how I chanced upon the dissertation topic. For that, I am grateful to George for getting me started on advancing his neat work on Translation Validation (the problem of checking equivalence of two programs), and to Alistair Sinclair for teaching me the first randomized algorithm in my life, that of polynomial identity testing (in his Algorithms class at Berkeley). I wondered if randomization can be used to check equivalence of two polynomials, then why not two programs.

Thanks are owed to a couple of other faculty members at Berkeley, besides George and Alistair. I want to thank Alex Aiken and Ras Bodik for providing me useful advice on various professional issues. I am grateful to Tom Henzinger for teaching me Model Checking, which was my most favorite class at Berkeley. I found Tom to be an amazing teacher; he could be very precise and formal and at the same time very abstract to appeal to one's intuition. I want to thank Bjorn Poonen for settling (the hardness of) some of the research related puzzles that I posed to him. He also did a wonderful job in teaching the Algebra class. Thanks to Ras and Bjorn (and of course, George) for their time and effort in serving on my dissertation committee and in providing useful feedback on a draft of this dissertation on a short notice.

I want to thank Jim Larus, Tom Ball, Trishul Chilimbi, and Sriram Rajamani at Microsoft Research, who always supported our research in numerous ways and made me feel valued. They also invited me for an internship in Redmond. Part of my stay at Berkeley was supported by a fellowship from Microsoft Research.

I want to thank Patrick Cousot for being appreciative of our research, which is simply a small addition to the large body of work on program analysis that has followed since the Cousots' seminal paper on abstract interpretation.

I want to thank Ashish Tiwari in whom I found a great friend and an excellent collaborator.

I want to thank all my fellow co-advisees and the larger OSQ crowd at Berkeley for the great atmosphere they provided for bouncing and discussing research ideas. I want to thank Jeremy Condit for grouping with me to work out the math problems in the Algebra course, which apparently is the hardest course that I ever took in my life. Rupak Majumdar was my college senior and I have always looked up to him and tried to emulate him in more ways than he might know. I have shamelessly bugged Ranjit Jhala for all sorts of questions, and he has always been kind enough to respond. Thanks to Tachio Terauchi, Dan Wilkerson, Matt Harren, Evan Chang, and Kamalika Chaudhuri for providing a lively atmosphere on the 5th floor of Soda Hall. Thanks to Helen Kang and La Shana for helping out with administrative issues in a seamless manner.

I want to thank all others who made Bay area such a special place to spend 5 golden years of my life: my college friends Prabhat Sinha, Rajnish Prasad, Ambarish Narayan, Apurv Gupta, Anwitaman Datta, Ambuj Tewari, Asha Tarachandani, Sourav Chatterjee, my college senior alumni Udai and Manju Singh, Rajesh and Kalpana Oberoi, and other friends Anurag Gupta, Gopal and Padma Gopinath Rao, Jai and Simi Balani, Nidhi Ajmani. I feel very fortunate to have spent a significant chunk of my time in the Bay area, which I consider to be one of the most beautiful place on Earth.

I will also like to take this opportunity to thank all my high school teachers and my alma mater professors who had a profound influence on me in my teenage years. Miss Leena Chatterjee, my English Literature high school teacher, apart from other things, taught us the real meaning of “education” (as derived from the word *educare*, which means to learn how to care). She is the most caring person I have known. Professor Y.N. Mohapatra at IIT Kanpur, besides teaching me physics, always spurred me to do something special. I can only hope to measure up to a small fraction of the expectations he has for me. Professor Dheeraj Sanghi was my undergraduate mentor, and Professor A.K. Sharma taught us interesting sociology classes.

I am highly indebted to my parents for always putting up to my whims without complaint, for the sacrifices they made to further our education, for their unwavering love and support, and for all those things that we took for granted. My dad, who used to teach math tuition to grade 10 students while himself in grade 8, has been a great source of inspiration for me. My maternal grandmother, whose wisdom has extended beyond her generation and has been a constant source of solace whenever I have felt low, deserves a special mention. Love of my sister Hanni, her husband Amit and other relatives have always kept me going.

Finally to Shivani, whom I can never thank enough. She always motivated me to reach for the stars. I did this all for her, and none of this would have been possible without her. I dedicate this dissertation to her.

That which is static and repetitive is boring. That which is dynamic and random is confusing. In between lies art.

John Locke.

Chapter 1

Introduction

This dissertation proposes a new program analysis technique called *random interpretation*. In this chapter, we first discuss some applications of program analysis followed by currently used design choices for program analysis. We then give an introduction to random interpretation, which exploits the sweet unexplored territory in the design space of program analysis leading to more efficient, more precise, and even simpler algorithms.

1.1 Program Analysis

Program analysis involves reasoning about programs in order to verify or discover their properties in an automatic manner.

Applications

Program analysis has applications in almost all areas of software development including program correctness, compiler optimizations, and translation validation.

As software increasingly dominates our lives, failures in software have a tendency to cause greater damage, including loss of lives. Studies estimate that software bugs cost businesses worldwide more than 200 billion dollars annually. This has lead to an increased focus on program analysis and verification tools. Programmers are increasingly using program analysis tools to find bugs in their programs, or to prove the absence of bugs. These may be low-level bugs like memory leaks, buffer overflows, or program-specific bugs like violation of certain data-structure invariants.

Apart from programmers, compilers need program analysis to generate optimized machine code. This allows programmers to program at a higher level of abstraction without worrying about low-level performance issues.

Another interesting application of program analysis is in translation validation, which is the problem of checking the semantic equivalence of programs before and after compilation [PSS98, Nec00]. The need for translation validation arises because compilers are complex pieces of software, and hence it is difficult to trust their output especially for safety-critical systems.

Design Choices

When constructing a program analysis to analyze a certain kind of program properties, there are several design parameters that can be chosen like completeness, computational complexity, and simplicity. Completeness of an analysis is a measure of how precise the analysis is (i.e., if some property holds of some program, then can the analysis reason about it?). Since any non-trivial program analysis is in general undecidable [Lan92], whatever analysis one designs, there is a class of programs that the analysis cannot precisely reason about, and it will report false positives (i.e., it will conservatively report that some property may not hold of a program, even though it did). The obvious trade-off that comes with increasing the precision of an analysis is that of increasing computational complexity (or running time) of the analysis. Another factor that is sometimes important in designing an analysis is the simplicity of the analysis in terms of its description or its implementation.

However, one factor that has remained constant in this design space is the soundness of an analysis, i.e., when an analysis claims that a certain property holds of a certain program, then it indeed is the case. It is interesting to consider if we can get any benefits by trading off some soundness. It turns out that by allowing soundness to be probabilistic, an analysis can be made more precise, more efficient and even simpler. By probabilistic soundness, we mean that if a property does not hold of a program, then with very high probability (over the random choices made by the analysis), the analysis will figure that out. In other words, there is a small probability that the analysis will erroneously announce that the property holds.

Such algorithms that have a small error probability are known as probabilistic algorithms or randomized algorithms. These algorithms have been successfully used in several areas of computer science [MR95, PRRR01] like: computational geometry [Mul00], simulation (Monte Carlo methods), network protocols (random standoff after contention), parallel computing (breaking Byzantine cases [CD89]), cryptography (generating keys, primality testing [AH87], digital signatures), proof-complexity theory (probabilistically checkable proofs [BGLR93]), etc. Even mathematical tools like Mathematica implement randomized algorithms. However, randomized algorithms have not been considered seriously in the area of program analysis and verification. We have improved the state-of-the-art for some program analysis problems (which had held its place for almost 30 years) with use of randomized algorithms, thereby demonstrating their potential in the area of program analysis. The algorithms that we have developed are based on a common theme that we call random interpretation.

1.2 Random Interpretation

Random interpretation can be seen as a combination of two complementary techniques for program analysis: random testing [Ham94] and abstract interpretation [CC77].

Random testing is a dynamic program analysis technique that simply involves testing a program on randomly chosen inputs. It is simple, efficient, and is quite commonly used to find bugs in programs. However, random testing is unsound because it usually cannot be used to prove absence of bugs in programs (since program inputs can potentially take an infinite number of values).

On the other hand, abstract interpretation is a static program analysis technique that is sound (and hence deterministic). This technique is so called because it involves interpreting (or analyzing) an abstraction (or approximation) of a program. Since any non-trivial program analysis is in general undecidable, any sound technique can only reason about an abstraction of a program; this is what abstract interpretation does. As the abstraction gets richer, the operations required to carry out abstract interpretation get more complicated and expensive. Examples of simple abstractions are signs abstraction [CC77] (which tracks signs of the values that program variables can take) and parity abstraction (which tracks parity of the values that program variables can take), while examples of more non-trivial abstractions are linear arithmetic abstraction [Kar76] and uninterpreted functions abstraction [RKS99].

Random interpretation is a new technique that combines the main ideas of these

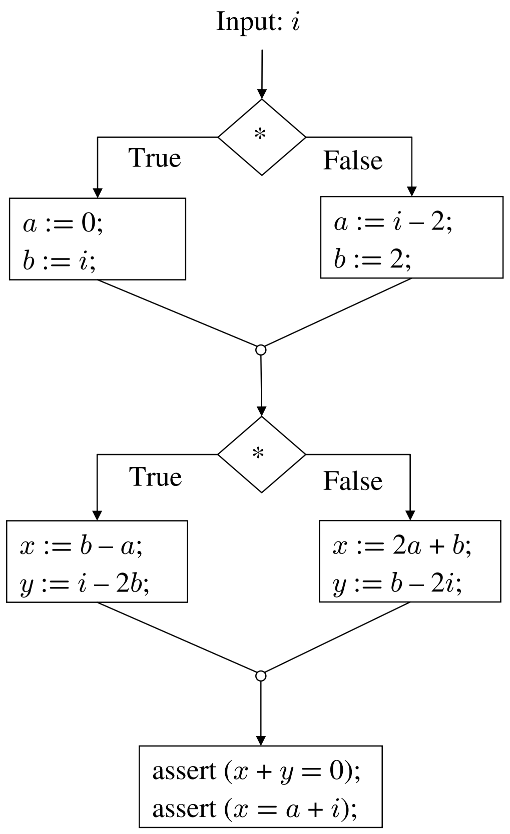


Figure 1.1: Example of a program with two assertions. The first assertion holds on all 4 paths, while the second one holds only on 3 paths (it does not hold when false branch is taken at the first conditional and true branch at the second conditional).

two complementary techniques and thus attempts to retain the strengths of both these techniques: simplicity and efficiency of random testing and soundness of abstract interpretation. Random interpretation is almost as simple and efficient as random testing, and has the advantage of offering better soundness guarantees. Random testing is not sound; it can only help find bugs in programs but it cannot prove absence of bugs. On the other hand, random interpretation can prove absence of bugs with very high probability. In that regard, random interpretation is almost as sound as abstract interpretation. Abstract interpretation is 100% sound, while the soundness of random interpretation can be made arbitrarily close to 100% by controlling various parameters of the algorithms.

We now explain these techniques in a little more detail by means of an example. Consider the program in Figure 1.1 represented as a flowchart. Note that the program has one input variable i. It has two conditionals, both of which have been abstracted away as non-deterministic branches, meaning that the conditional can non-deterministically evaluate to either true or false irrespective of the program state before it. Thus, there are 4 execution paths in this program. The program has two assertions at the end. The first assertion is true since it holds on all the 4 paths in the program. The second assertion is false because there is at least 1 path in the program on which it fails, in particular when the first branch evaluates to false and the second branch evaluate to true. In fact, this path is the only path in the program on which the assertion fails.

If one were to use random testing to decide the validity of these assertions, then it is imperative that the execution path that violates the second assertion is tested. If the execution of all paths is equally likely, then the probability of testing the violating path for the second assertion is $\frac{1}{4}$. In general, a program with n such diamond structures has 2^n paths, and there may be exactly one violating path for a given false assertion; in that case the probability of testing the violating path will be $\frac{1}{2^n}$, which is very small. Hence, quite likely, random testing cannot detect all violating assertions, and hence is unsound.

On the other hand, we can (successfully) use abstract interpretation based techniques to decide the validity of these assertions. This involves performing a forward analysis on the program and computing the invariant at each program point from the invariants at the preceding program points. In this case, these invariants are sets of linear equalities among program variables. Figure 1.2 shows the program in Figure 1.1 annotated with such invariants. Note that the invariant at the start of the program is simply the empty set of linear equalities. This reflects that we do not assume anything about the input variables of the program, and they can take any value. The invariants at program points π_1 and π_2 are represented by the sets $\{a{=}0,\ b{=}i\}$ and $\{a{=}i{-}2,\ b{=}2\}$ respectively. These seem easy to compute but in general this operation of computing the strongest post-condition in an abstraction is non-trivial. The invariant at program point π_3 is the strongest set of linear equalities that are implied by the set of linear equalities at points π_1 and π_2. The set $\{a{+}b{=}i\}$ represents all such equalities. An operation that performs such a computation is called the join operation and is usually the hardest operation in an abstract interpretation based program analysis. Similarly, note that the join of the set of linear equalities at program points π_4 and π_5 is represented by the set $\{a{+}b{=}i,\ x{=}{-}y\}$. Since $\{a{+}b{=}i,\ x{=}{-}y\}$ implies $x + y = 0$, we claim that the first assertion holds, and since $\{a{+}b{=}i,\ x{=}{-}y\}$ does not imply $x = a + i$, we claim that the second assertion may not hold.

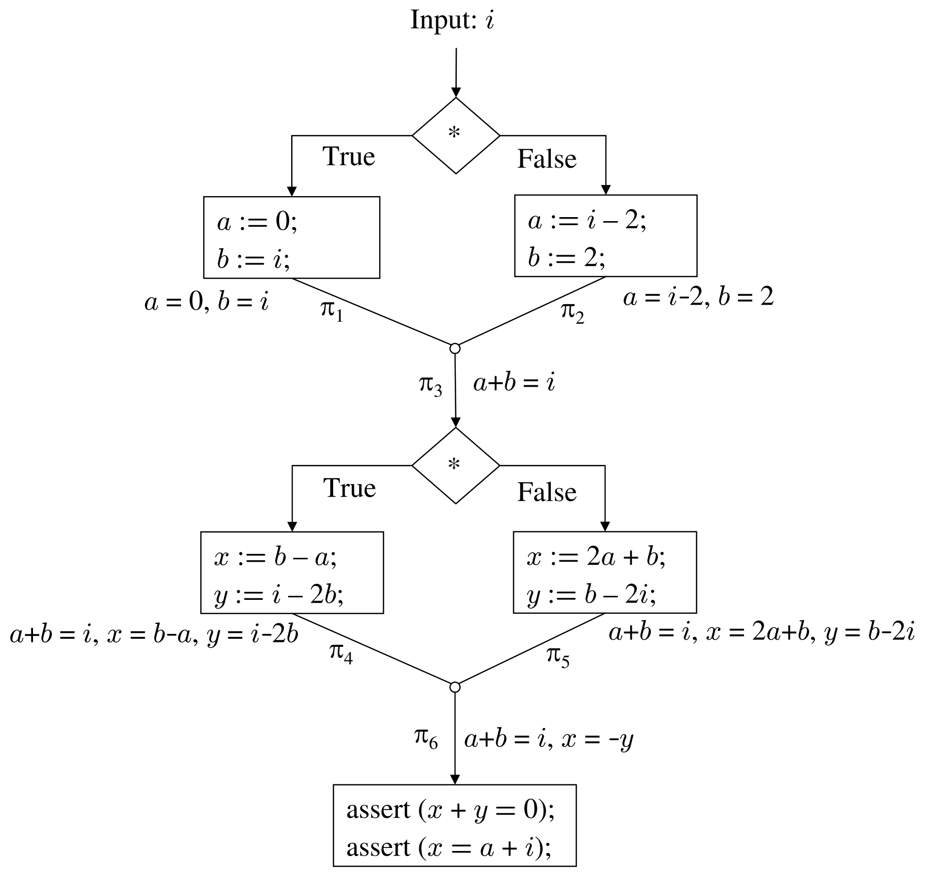


Figure 1.2: Deciding validity of linear equality assertions using abstract interpretation. The sets of linear equalities shown next to each edge represent the invariants computed by the abstract interpreter.

Our random interpretation based technique can also decide the validity of the two assertions at the end of the program in Figure 1.1 like abstract interpretation; but unlike abstract interpretation it maintains the simplicity and efficiency of random testing. Any random interpretation based algorithm (called random interpreter) has the following generic skeleton.

- Choose random values for the input variables.
- Execute both branches of conditionals.
- Combine values of variables at join points.
- Test the assertion.

A random interpreter essentially executes the program in a non-standard manner. It first chooses random values for the input variables of the program, just like random testing. It then starts executing the program. When it reaches a conditional, then instead of evaluating the conditional and figuring out whether it is true or false, and accordingly taking either the true branch or the false branch, the random interpreter executes both branches of the conditional (in parallel), just like an abstract interpreter. When the random interpreter approaches a join point, then there are two program states, one on either side of the join point. The random interpreter combines those program states ρ_1 and ρ_2 by means of an operation called *affine join* operation to generate another program state ρ. The affine join operation is a simple operation that involves taking a random weighted average of the two states to be combined; this operation is further discussed in Section 2.1.1. The combined state ρ is a hypothetical state in the sense that it may never arise in any real execution of the program. However, it has the useful property that it satisfies exactly those interesting properties (in this case, linear equalities) that are satisfied by both ρ_1 and ρ_2 with high probability over the choice of the random weight used by the random interpreter for computing the affine join. Hence, instead of proceeding the execution with both states ρ_1 and ρ_2, it suffices to continue the execution with simply the state ρ. (Note that it is important to proceed the execution from the join point with only one program state; because otherwise if the random interpreter works with both states ρ_1 and ρ_2, then at the next join point there will be 4 program states; and at the next one, there will be 8 states, and so on. This is like testing all execution paths in parallel and there can be potentially exponential number of paths even in a loop-free program.) Once the random interpreter is done executing the program in this non-standard manner, then it simply tests the assertions at a program point with the program state at that point, just like random testing. In Section 2.1.1, we show in full detail how the random interpreter decides the validity of the two assertions at the end of the program in Figure 1.1 using the affine join operation.

Hence, we see that a random interpreter attempts to retain the simplicity and efficiency of random testing (by simply executing the program, and combining program states at join points), and soundness of abstract interpretation (by executing both branches of conditionals). This comes at the cost of probabilistic soundness, meaning that there is a chance that the random interpreter gives a false judgment. This happens when the random interpreter makes some bad random choices. However, the good thing is that the number of such bad random choices is small and if the random choices are made from a large set, then the probability that the random interpreter yields a false judgment can be made arbitrarily small. It is also important that this probability is not over the space of all programs, but over the space of random choices.

In the following chapters, we describe formally some program abstractions to which this approach can be applied and the kind of properties that can be verified and discovered of those programs. The reader will notice that the key ideas in random interpretation are simple to describe but the error probability analysis is the most challenging part (as is typical of probabilistic algorithms).

Organization

This dissertation is organized as follows. We first describe a random interpretation based program analysis for the abstraction of linear arithmetic in Chapter 2. It can be used to reason about linear equalities between program variables. Chapter 3 describes another random interpretation based program analysis for the abstraction of uninterpreted functions. It can be used to reason about Herbrand equivalences between program subexpressions (global value numbering). In Chapter 4, we describe a framework that generalizes the random interpreters described in Chapter 2 and Chapter 3. We then show how to extend any random interpreter described using this framework to an inter-procedural setting to obtain a more precise analysis. This framework should guide the development of random interpreters for new domains, and provide a large part of their error probability analysis. Chapter 5 discusses the problem of combining the random interpreters from Chapter 2 and Chapter 3 to build a more precise program analysis for the combined abstraction of linear arithmetic and uninterpreted functions.

In some very real sense, computation is inherently randomized. It can be argued that the probability that a computer will be destroyed by a meteorite during any given microsecond of its operation is at least 2^{-100}.

Christos Papadimitriou, *Computational Complexity.*

Chapter 2

Linear Arithmetic

In this chapter, we consider the problem of verifying or discovering linear equalities among program variables at any program point. A linear equality among variables $x_1, \ldots, x_k$ is a relationship of the form $c_0 + \sum_{i=1}^{k} c_i x_i = 0$, where c_i's are some constants. We consider programs that have been abstracted using linear assignments (i.e., assignments of the form $x := e$, where e is a linear arithmetic expression) and linear equality conditionals (i.e., conditionals of the form $e = 0$). This program abstraction is described more formally in Section 2.2. We assume that the program variables take rational values. It turns out that the problem of discovering all linear equalities even in such programs is undecidable [MOS04a]. We present a polynomial time algorithm that discovers (some) linear equalities for such programs [GN03]. (This algorithm discovers all linear equalities that hold when all conditionals in the program are treated as non-deterministic. It also discovers some linear equalities that require reasoning about predicates in conditionals.)

Applications: Several classical data flow analysis problems can be modeled as the problem of detecting linear equalities among program variables. Examples are: constant propagation (which involves detecting variables that always have constant values), copy propagation (which involves detecting variables that always have same values), and common subexpression elimination. Several loop invariant computations and loop induction variables can also be identified by detecting linear equalities. Translation validation [PSS98, Nec00] also requires checking the equality of variables in two versions of a program before and after optimization.

2.1 Key Ideas

In this section, we illustrate the two key ideas in our random interpretation based algorithm for discovering linear equalities among program variables.

The random interpreter performs arithmetic over the field $\mathbb{F}_p$ for some randomly chosen prime p (as opposed to the field of rational numbers) to avoid the problem of dealing with big numbers. $\mathbb{F}_p$ denotes the field of integers $\{0, \ldots, p-1\}$ where arithmetic is done modulo p. Whenever the random interpreter chooses some random value, it does so independently of the previous choices and uniformly at random (u.a.r.) from the field $\mathbb{F}_p$. We use `Rand`() to denote such a fresh random value.

2.1.1 Affine Join Operation

The random interpreter executes a program in a non-standard manner. When it reaches a conditional, it executes both its branches. As a result, when it reaches a join point, there are two program states ρ_1 and ρ_2, one on either side of the join point. The random interpreter combines the two program states ρ_1 and ρ_2 using an operation called *affine join* operation.

A program state is a mapping from program variables that are visible at the corresponding program point to values, which in this case are elements of $\mathbb{F}_p$. In general these values are polynomials over $\mathbb{F}_p$. These polynomials may simply be elements of $\mathbb{F}_p$ (as in this chapter), vectors of elements from $\mathbb{F}_p$ [1] (Chapter 3), or linear functions of the program's input variables (Chapter 4). We first define the affine join operation for combining two values, and then extend it to combine two program states.

The affine join operation for combining two values v_1 and v_2 involves computing the weighted average of v_1 and v_2 using the weights w and $1-w$, where w is chosen randomly from $\mathbb{F}_p$. We denote this operation by ϕ_w.

$$\phi_w(v_1, v_2) = wv_1 + (1-w)v_2$$

[1]A vector $(v_1, \ldots, v_\ell)$ can be represented by the polynomial $\sum_{i=1}^{\ell} z_i v_i$, where $z_1, \ldots, z_\ell$ are some fresh variables.

ϕ_w can be treated as some kind of selector between v_1 and v_2. Note that if w is chosen to be either 1 or 0, then the affine join operation yields either v_1 or v_2 respectively.

$$\phi_1(v_1, v_2) = v_1 \quad \text{and} \quad \phi_0(v_1, v_2) = v_2$$

However, the interesting bit lies in plugging a non-boolean value for w, in fact, a random value for w, which has the effect of producing some (random) combination or superposition of v_1 and v_2 rather than simply v_1 or v_2.

The affine join operation for combining values as described above can be extended to combine program states ρ_1 and ρ_2 as follows.

$$\phi_w(\rho_1, \rho_2)(x) = \phi_w(\rho_1(x), \rho_2(x))$$

The affine join of states ρ_1 and ρ_2 is another state ρ that assigns to each variable x the affine join of the values of variable x in the states ρ_1 and ρ_2. Note that the same random weight w is used for combining the two values of all variables. For example, let ρ_1 and ρ_2 be the following states of two variables x and y.

$$\rho_1 : \{x \rightarrow 2, y \rightarrow 3\} \text{ and } \rho_2 : \{x \rightarrow 4, y \rightarrow 1\}$$

The affine join of ρ_1 and ρ_2 with respect to the random weight, say 3, is the following state ρ:

$$\rho = \phi_3(\rho_1, \rho_2) : \{x \rightarrow \phi_3(2, 4) = -2, \;\; y \rightarrow \phi_3(3, 1) = 7\}$$

Any state ρ obtained as the affine join of two states ρ_1 and ρ_2 has two interesting properties: (a) The state ρ satisfies all linear equalities that are satisfied by both ρ_1 and ρ_2. In the above example, note that both ρ_1 and ρ_2 satisfy the linear equality $x + y = 5$, and hence ρ also satisfies it. (b) On the other hand, given a linear equality that is not satisfied by at least one of ρ_1 or ρ_2, ρ will also not satisfy that linear equality with very high probability (over the choice of the random weight w used to compute the affine join). In the above example, note that $y = 2$ is a linear equality that is not satisfied by at least one of ρ_1 and ρ_2 (in fact, it is not satisfied by both). A simple calculation will show that the probability that ρ will satisfy that linear equality is same as the probability of choosing w to be $\frac{1}{2}$. Lemma 1 and Lemma 2 state and prove slightly more general versions of these two properties. Hence, in some sense, the affine join of states ρ_1 and ρ_2 filters the common linear equalities that are satisfied by both the states ρ_1 and ρ_2.

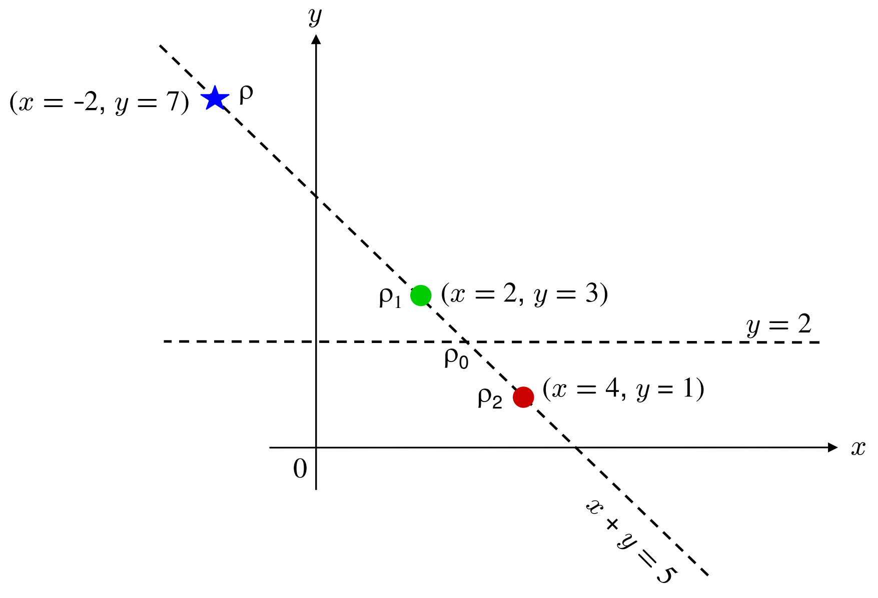


Figure 2.1: Geometric interpretation of the affine join operation.

It is worth-pointing out that, unfortunately, the affine join operation does not filter common non-linear relationships. In our example above, note that both states ρ_1 and ρ_2 satisfy the invariant $x(1+y) = 8$, but ρ does not satisfy this invariant. This fact will not have any bearing on the algorithm that we describe in this chapter for analyzing a program for linear equalities. However, as we will notice in Section 3.1, this fact needs to be given attention to when we try to use similar approach to analyze properties of different kinds of abstractions of a program.

The affine join operation has a nice geometric interpretation. A program state of k variables geometrically represents a point in a k-dimensional space. The affine join of two such points ρ_1 and ρ_2 corresponds to drawing a line between those points and choosing a point ρ randomly on that line. Figure 2.1 shows the geometric interpretation of the affine join operation for the above example. The two interesting properties of the affine join can also be explained geometrically: (a) Any linear equality that is satisfied by both ρ_1 and ρ_2 is represented geometrically by a hyperplane that passes through both ρ_1 and ρ_2, and thus also through the line joining ρ_1 and ρ_2. Since ρ lies on the line joining ρ_1 and ρ_2, the hyperplane also passes through the point ρ, and hence ρ also satisfies the corresponding linear equality that the hyperplane represents. This explains why we choose point ρ on the

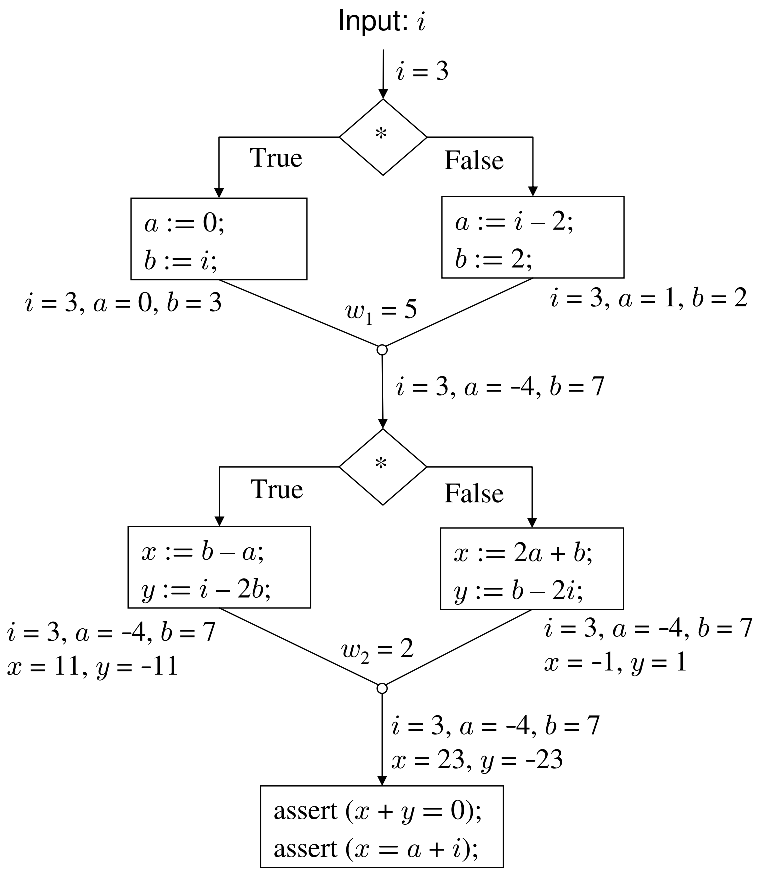


Figure 2.2: Deciding validity of linear equality assertions using random interpretation. The values of variables shown next to each edge represent the program states computed in a random interpretation.

line joining ρ_1 and ρ_2. (b) Any linear equality $e = 0$ that is not satisfied by at least one of the points ρ_1 and ρ_2 is represented geometrically by a hyperplane that does not pass through at least one of ρ_1 and ρ_2, and thus also does not contain the line joining ρ_1 and ρ_2. In fact, the hyperplane $e = 0$ intersects the line joining ρ_1 and ρ_2 in at most one point, say ρ_0. The probability that ρ satisfies the linear equality $e = 0$ is same as the probability that we pick ρ to be ρ_0 during the process of choosing a point randomly on the line joining ρ_1 and ρ_2. The probability of this event is extremely small since there are many points on the line joining ρ_1 and ρ_2. This explains why we choose the point ρ randomly. In this chapter, we occasionally refer to states, when used in a geometric context, as points.

We now show how the random interpreter uses this affine join operation to decide the validity of the two assertions in the program shown in Figure 2.2. The random interpreter starts with a random value, say 3, for the input variable i and then executes the assignment statements on both sides of the conditional. In the example, we show the values of all live variables at each program point. The two program states before the first join point are combined with the affine join operation using the random weight $w_1 = 5$. Note that the resulting program state after the first join point can never arise in any real execution of the program. However, this state captures the invariant $a + b = i$, which is necessary to prove the first assertion in the program. Also, note that the random interpreter does not compute the invariant $a + b = i$ symbolically (unlike the abstract interpreter described in Section 1.2); but this invariant is neatly hidden in the number game that is being played here. The random interpreter then executes both sides of the second conditional and computes the affine join of the two states before the second join point using the random weight $w_2 = 2$. We can verify easily that the resulting state at the end of the program satisfies the first assertion but does not satisfy the second. The random interpreter thus claims that the first assertion is true, and the second assertion may not be true. Notice that in one (non-standard) run of the program the random interpreter has figured out that one of the (potentially) exponentially many paths violates the second assertion.

Note that choosing w to be either 0 or 1 at a join point corresponds to executing either the true branch or the false branch of the corresponding conditional; this is what naive testing accomplishes. However, by choosing w (randomly) from a set that also contains non-boolean values, the random interpreter is able to capture the effect of both branches of a conditional in just one interpretation of the program. In fact, there is nothing particular about the random interpreter's specific choice of $w_1 = 5$ and $w_2 = 2$. Whatever values the random interpreter would have picked for w_1 and w_2, it would have been able to verify the first assertion. On the other hand, for almost all values of w_1 and w_2, the random interpreter would have been able to conclude that the second assertion is false. There are a few bad random choices for i, w_1 and w_2 (namely, $i = 2$, or $w_1 = 1$, or $w_2 = 0$) and if the random interpreter would have chosen those values, then it would have incorrectly validated the second assertion. However, since the random interpreter chooses these values from $\mathbb{F}_p$, which is a large enough set, the probability of hitting any of those bad random choices is extremely small, in this case at most $\frac{3}{p}$, even if the random interpreter does not know what those bad random choices are. (Note that even though in this case it turns out that any non-boolean choice for the weights is good, this may not be true always. In general, there exists at most one bad random weight at each join point that may mislead the random interpreter into validating an incorrect assertion.)

What we have described above is one run of the random interpreter over a program. There are several advantages in performing multiple such runs of the program, wherein the random interpreter declares an assertion to be true iff the assertion is true on all those runs. This reduces the error probability by a factor that is exponential in the number of runs. Another advantage lies in being able to discover linear equalities rather than simply verifying them. Note that one run is good enough for verifying a given assertion. However, if the random interpreter attempts to discover linear equalities by analyzing the values of the variables in just one run, then it may draw incorrect conclusions. For example, from the state computed by the random interpreter at the end of the program in Figure 2.2, it may (incorrectly) appear that $a = -4$ is an invariant at the end of the program. This problem can be avoided if the random interpreter performs several such runs of the program and then looks for common relationships among all those executions. A close analysis of our example shows that if the random interpreter executes the program once more, the probability of a evaluating to -4 again is precisely the probability that we choose the random weight w_1 to be 5 again, which is very small. Section 2.1.2 describes another advantage of performing several runs, wherein they are used to infer linear equalities that require reasoning about predicates in the conditionals (as opposed to abstracting the conditionals away as non-deterministic, which is what we have done here).

Multiple runs of the random interpreter over a program yield a collection of, say t, states at each program point. We refer to this collection of states as a sample. We use the notation S_i to refer to the i^{th} state in sample S. We extend the affine join operation to combine two samples, in which case we combine each pair of corresponding states using a separate weight factor. Hence, the affine combination of two samples S and S' is another sample denoted by $\phi_{[w_1,\ldots,w_t]}(S, S')$, where:

$$\phi_{[w_1,\ldots,w_t]}(S, S')_i = \phi_{w_i}(S_i, S'_i) \qquad \text{for } 1 \le i \le t$$

A sample can be thought of as a representation of the affine subspace defined by the states in the sample. An affine join of two samples in that sense computes the union of two subspaces (in a very efficient manner). On the other hand, computing union of two subspaces when represented by symbolic linear equalities is an involved operation, and this is what the abstract interpretation based algorithm for discovering linear equalities does [Kar76].

We conclude the discussion of the affine join operation with the statement and proof of the completeness and probabilistic soundness lemmas. We use the notation $\texttt{Eval}(e, \rho)$ to denote the standard meaning of expression e in state ρ. A more formal definition is given in Section 2.2.1. We say that a state ρ satisfies an equality $e_1 = e_2$, denoted by $\rho \models e_1 = e_2$, iff $\texttt{Eval}(e_1, \rho) = \texttt{Eval}(e_2, \rho)$. We extend this notion to samples and say a sample S satisfies an equality $e_1 = e_2$ when all of its states satisfy the equality. We denote this by $S \models e_1 = e_2$.

The completeness lemma for the affine join operation states that the resulting sample satisfies all linear equalities that are satisfied by both the original states.

Lemma 1 [Affine Join Completeness Lemma] *Let S^1 and S^2 be two t-state samples and let g be an expression such that $S^1 \models g = 0$ and $S^2 \models g = 0$. Then, for any choice of weights $w_1, \ldots, w_t$, the affine join $S = \phi_{[w_1,\ldots,w_t]}(S^1, S^2)$ is such that $S \models g = 0$.*

Proof. It is easy to verify that for any affine combination of two states $\phi_w(\rho_1, \rho_2)$, we have $\texttt{Eval}(g, \phi_w(\rho_1, \rho_2)) = \phi_w(\texttt{Eval}(g, \rho_1), \texttt{Eval}(g, \rho_2))$, since g is a linear expression. Thus if the value of g is zero in the states ρ_1 and ρ_2, then the value of g is also zero in their affine combination. From here the completeness statement follows immediately. $\square$

The following probabilistic soundness lemma for the affine join operation states that the probability of choosing the combination weights such that a given false linear equality (i.e., a linear equality that is not satisfied by at least one of the two original states) is introduced is extremely small (for a large enough choice of weights). The notation $\Pr(\mathcal{E})$ denotes the probability of event $\mathcal{E}$ over the corresponding random choices.

Lemma 2 [Affine Join Soundness Lemma] *Let S^1 and S^2 be two samples and let g be an expression such that $S^1 \not\models g = 0$. More specifically, let ℓ be the number of states in S^1 that do not satisfy $g = 0$. Let $w_1, \ldots, w_t$ be chosen u.a.r. from $\mathbb{F}_p$ and independently of each other and of the expression g. Let $S = \phi_{[w_1,\ldots,w_t]}(S^1, S^2)$. Then, $\Pr(S \models g = 0) \le \left(\frac{1}{p}\right)^{\ell}$.*

Proof. Without any loss of generality, let $S^1_1, \ldots, S^1_\ell$ be the ℓ states in S^1 that do not satisfy $g = 0$. For any $i \in \{1, \ldots, \ell\}$, consider the line joining the points S^1_i and S^2_i. If $\texttt{Eval}(g, S^1_i) = \texttt{Eval}(g, S^2_i)$, then this line is parallel to the hyperplane $g = 0$, and hence, no point on this line satisfies the equation $g = 0$. In other words, for any choice of w_i, $\texttt{Eval}(g, S_i) = \texttt{Eval}(g, S^1_i) = \texttt{Eval}(g, S^2_i) \neq 0$ and thus the probability that $S_i \models g = 0$ is zero. If $\texttt{Eval}(g, S^1_i) \neq \texttt{Eval}(g, S^2_i)$, then this line intersects the hyperplane $g = 0$ in exactly

one point. In other words, there is only one choice for w_i (namely, $\frac{\texttt{Eval}(g,S_i^2)}{\texttt{Eval}(g,S_i^2)-\texttt{Eval}(g,S_i^1)}$) such that $\texttt{Eval}(g, S_i) = 0$. Thus, the probability that the weight w_i is chosen such that S_i satisfies the equation $g = 0$ is precisely $\frac{1}{p}$. Since $w_1, \ldots, w_\ell$ are all independent, it follows that the probability that all the states $S_1, \ldots, S_\ell$ satisfy the equation $g = 0$ is at most $\left(\frac{1}{p}\right)^\ell$.
$\square$

2.1.2 Adjust Operation

In the example program in Figure 2.2 conditionals were abstracted away as non-deterministic branches. In this section, we describe how to infer linear equalities that require reasoning about predicates in the conditionals.

Consider the following program:

```
a := x + y ;
if x = y then b := a else b := 2x ;
assert (b = 2x);
```

The assert statement at the end of the above program is true but in order to prove it we must notice that $x = y$ in the true branch of the conditional. The state computed by the random interpreter before the conditional quite likely will not satisfy the predicate $x = y$ in the conditional. In that case, if the random interpreter proceeds with the same state on the true branch of the conditional, then it is not capturing the extra information that is provided by the conditional on that branch, namely that the predicate $x = y$ is true. We could try to restart the interpretation with values of the input variables x and y that satisfy the conditional, but finding such initial values in general is hard. Also, notice that we could not do something simple like replace the occurrences of x with y in the true branch; this would not help in this case. We have to somehow adjust all of the previously computed variables as well, such as a in this example.

To solve this problem, we propose that the random interpreter performs multiple executions of the program in parallel, thereby computing several states, which we refer to as a sample, at each program point. We now describe an operation for transforming a sample in such a way that the new sample satisfies all the linear equalities that are satisfied by the original sample, and additionally it satisfies the linear equality given by the conditional. We do this by essentially “projecting” the points in the sample onto the hyperplane represented

```
Adjust($S, e$) =
1 Pick $S_{j_1}$ and $S_{j_2}$ from $S$ such that Eval($e, S_{j_1}$) $\neq$ Eval($e, S_{j_2}$).
2 Pick $w \in \mathbb{F}_p$ u.a.r. with the constraint that $\rho_0 = \phi_w(S_{j_1}, S_{j_2})$ is such that
          Eval($e, \rho_0$) $\neq 0$ and Eval($e, \rho_0$) $\neq$ Eval($e, S_i$) for all $i \in \{1, .., t\}$.
3 For $i \in \{1, \ldots, t\}$ do
      Let $S''_i$ be the intersection of hyperplane $e = 0$ with the line passing
      through $S_i$ and $\rho_0$, i.e., $S''_i = \phi_w(S_i, \rho_0)$ for $w = \frac{\texttt{Eval}(e,\rho_0)}{\texttt{Eval}(e,\rho_0)-\texttt{Eval}(e,S_i)}$.
4 For $i \in \{1, \ldots, t\} - \{j_1, j_2\}$ do
      Let $S'_i = S''_i$.
5 For $i \in \{j_1, j_2\}$ do
      Pick $w_{i,1}, \ldots, w_{i,t} \in \mathbb{F}_p$ independently and u.a.r. with the constraint
      that $\sum_{j=1}^{t} w_{i,j} = 1$.
      Let $S'_i$ be such that $S'_i(x) = \sum_{j=1}^{t} w_{i,j}\ S''_j(x)$ for all variables $x$.
6 Return $[S'_1, \ldots, S'_t]$.
```

Figure 2.3: The Adjust operation.

by the conditional. Orthogonal projection does not work since it destroys linear equalities that are satisfied by S. Instead we use the function Adjust(S, e) described in Figure 2.3 to adjust the sample S according to the conditional $e = 0$.

There are a couple of details in the definition of the Adjust operation that deserve discussion. Line 1 presumes the existence of two states S_{j_1} and S_{j_2} in which the expression e has different values. If there are no such states, it means that the expression e has same value, say c, in all states in the sample. In such a case we need not perform any adjustment. Instead we declare that the linear equality $e = c$ holds before the conditional and accordingly we consider only one branch depending on whether the constant c is zero or not. When this is the case we say that Adjust(S, e) is undefined.

Line 2 in the Adjust function chooses a point ρ_0 u.a.r. on the line passing through S_{j_1} and S_{j_2} with the constraint that ρ_0 does not lie on the hyperplane $e = 0$ (i.e., Eval(e, ρ_0) $\neq 0$) and ρ_0 is at different distance from the hyperplane $e = 0$ compared to the distance of any point S_i in the sample S (i.e., Eval(e, ρ_0) $\neq$ Eval(e, S_i)). Since e has different values

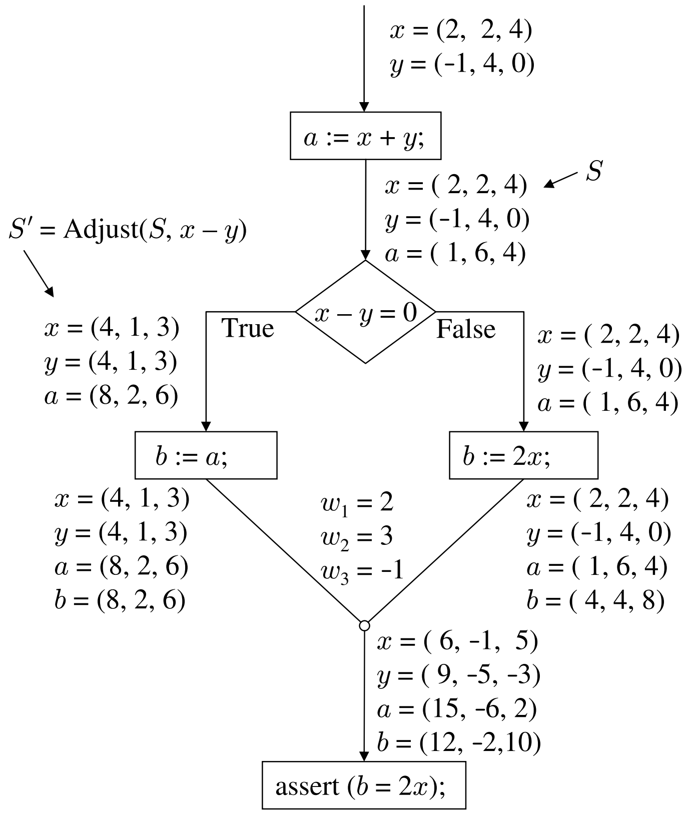


Figure 2.4: Use of the Adjust operation to decide validity of linear equality assertions. The 3 values for variables shown next to each edge represent 3-state samples computed by the random interpreter. The Adjust operation is used to obtain the sample S' from S, as detailed in Figure 2.5.

at S_{j_1} and S_{j_2}, such a point ρ_0 always exists (in fact, there are at least $p-(1+t)$ choices for ρ_0), and choosing such a point is a linear-time operation. The loop on line 3 involves computing the intersection of the line that passes through ρ_0 and S_i with the hyperplane $e=0$ (this is a simple computation). Note that because of the nature of choice of ρ_0, the line passing through ρ_0 and S_i indeed intersects the hyperplane $e=0$.

The loop on line 4 assigns points $S'_i = S''_i$ (for $i \neq j_1, j_2$), while the loop on line 5 assigns points S'_{j_1} and S'_{j_2} a random affine combination (i.e., weighted average with sum of the weights being 1) of all points in the sample S''. The intuition behind doing this is as follows. Note that S''_{j_1} and S''_{j_2} as computed in the loop on line 3 are identical points, and hence not independent of each other. Independence of these points is crucial for proving the probabilistic soundness of the algorithm. The affine combination of points in the sample S''

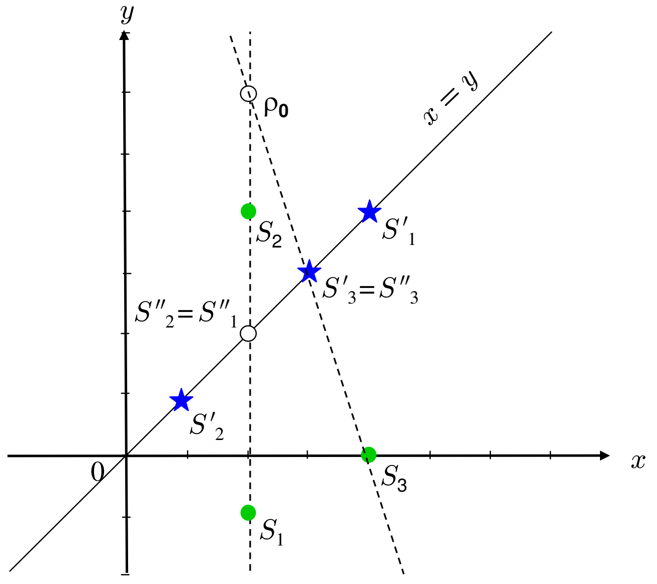


Figure 2.5: Geometric interpretation of the Adjust operation. This also shows in detail how the sample S in Figure 2.4 is adjusted to obtain sample S'.

generates fresh independent points in the subspace that is spanned by points in the sample S''.

Note that all states in the sample S' are obtained as some affine combinations of the states in the sample S. Hence, they satisfy all linear equalities between variables that are satisfied by all states in the original sample S. Furthermore, it should be intuitive that if the states in the sample S are spread in some subspace U, then the states in the adjusted sample S' are spread in the intersection of the subspace U with the hyperplane $e = 0$.

We now illustrate the use of the Adjust operation to verify the assertion in the program mentioned at the beginning of this section. Figure 2.4 shows the program executed on the 3-state sample shown at the top of the figure. The Adjust operation is used to obtain the sample S' from S as follows. Notice that all states in S satisfy $a = x + y$ (due to the assignment). Now consider the distribution of the points in S when viewed inside the hyperplane $a = x + y$ as shown in Figure 2.5. We pick the points S_1 and S_2 to play the role of S_{j_1} and S_{j_2} from the definition of `Adjust` (since the expression $x - y$ has different values on those points). Then we pick another point ρ_0 ($\{x \to 2, y \to 6\}$) u.a.r. on the line determined by S_1 and S_2 (which incidentally is parallel to the y-axis) with the constraint that ρ_0 is not in the hyperplane $x - y = 0$, and that the lines passing through ρ_0 and any point in S are not parallel to the hyperplane $x - y = 0$. Then we obtain the points S''_i (for $i = 1, 2, 3$) as the intersections of the lines that pass through ρ_0 and S_i with the hyperplane $x - y = 0$. Notice that two of the points S''_1 and S''_2 coincide. We assign $S'_3 = S''_3$ while we obtain S'_1 and S'_2 by generating two fresh points by taking affine combinations of the points in the sample S''.

Notice that the sample before the equality conditional is adjusted only in its true branch. For the false branch, the random interpreter uses the same sample before the conditional since there is no linear equality that can be inferred from a disequality. Notice that after adjustment the sample satisfies both the original relationships ($a = x + y$) and also the new one due to the conditional ($x = y$). Finally, the affine join operation is done at the join point using the random weights 2, 3, and -1 and the resulting sample now clearly reflects the desired assertion $b = 2x$ (precisely because both sides of the join reflect the same assertion).

To complete the discussion of the Adjust operation, we state and prove below the completeness and soundness lemmas. The completeness lemma states that the adjusted sample satisfies all the linear equalities satisfied by the original sample and additionally also satisfies the linear equality for which the original sample was adjusted.

Lemma 3 [Adjust Completeness Lemma] *Let e be any expression and S be any sample such that $S' = \texttt{Adjust}(S, e)$ is defined. Then, $S' \models e = 0$. Furthermore, for any expression g, if $S \models g = 0$ then $S' \models g = 0$.*

Proof. By definition of the sample S', we have that each state S'_i from S' satisfies $e = 0$. Let ρ_0 be the intermediate state obtained as an affine combination of S_{j_1} and S_{j_2} (in line 2 of the `Adjust` function). Clearly ρ_0 satisfies all linear equalities that both S_{j_1} and S_{j_2} satisfy, hence also $g = 0$. Now each S'_i from S' is an affine combination of S_i and ρ_0 and therefore it also satisfies $g = 0$. $\square$

The following soundness lemma implies that the adjusted sample satisfies exactly one more linearly independent linear equality than the original sample.

Lemma 4 [Adjust Soundness Lemma] *Let e be any expression and S be any sample such that $S' = \texttt{Adjust}(S, e)$ is defined. Let S'' be the sample computed by the Adjust operation immediately after the loop in line 3. Let ρ_0 be the intermediate state computed in line 2 of the `Adjust` function. For any expression g, there exists a constant c such that if* *any ℓ states in S'' satisfy the linear equality $g = 0$, then the corresponding ℓ states in the sample S, as well as the state ρ_0, satisfy the linear equality $g + ce = 0$.*

Proof. Let c be $-\frac{\texttt{Eval}(g,\rho_0)}{\texttt{Eval}(e,\rho_0)}$. Note that ρ_0 satisfies $g + ce = 0$. Without any loss of generality, let $S''_1, \ldots, S''_\ell$ be the ℓ states in sample S'' that satisfy $g = 0$. Consider any $i \in \{1, \ldots, \ell\}$. Note that S''_i satisfies $g + ce = 0$ since S''_i satisfies both $g = 0$ and $e = 0$. Since $S''_i = \phi_{w_i}(S_i, \rho_0)$ (for $w_i = \frac{\texttt{Eval}(e,\rho_0)}{\texttt{Eval}(e,\rho_0)-\texttt{Eval}(e,S_i)}$), it can be easily verified that S_i also satisfies $g + ce = 0$. □

The geometric intuition behind the soundness lemma is that if some subset of the adjusted points in the sample S'' (computed in the loop in line 3) lie in some hyperplane $g = 0$, then the corresponding subset of the original points lie in the hyperplane that contains the point ρ_0 and passes through the intersection of the hyperplanes $g = 0$ and $e = 0$. The soundness lemma implies that any equation $g = 0$ that is satisfied by the adjusted sample can be expressed as a linear combination of the linear equality $e = 0$ and some linear equality $g' = 0$ that is satisfied by the original sample. Note that the soundness lemma indicates such a g' (namely, $g + ce$).

2.2 The Random Interpreter

We first formally describe our program model and some notation related to it.

Program Model

We assume that each procedure in a program is abstracted using the flowchart nodes shown in Figure 2.6. In the assignment node, x refers to a program variable while e denotes a linear arithmetic expression:

$$e ::= x \quad | \quad c \quad | \quad e_1 + e_2 \quad | \quad e_1 - e_2 \quad | \quad c \times e$$

Here c denotes some arithmetic constant. We express the computational complexity of algorithms in terms of the number of assignment nodes and for that purpose, we assume that the expression e in an assignment node is of constant size. A non-deterministic assignment $x :=?$ denotes that the variable x can be assigned any value. Such non-deterministic assignments are used as a safe abstraction of statements (in the original source program) that our abstraction cannot handle precisely.

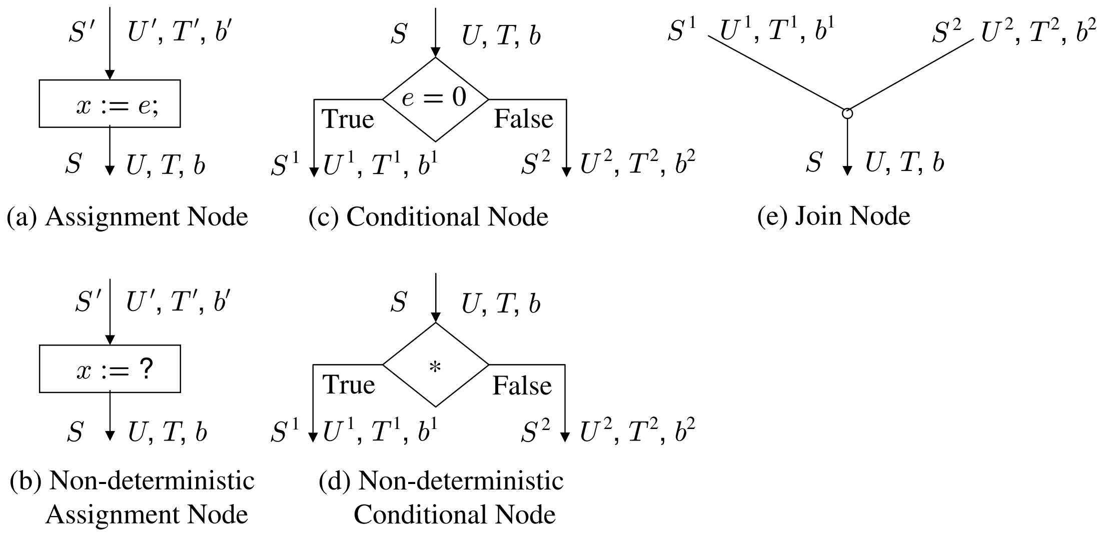


Figure 2.6: Flowchart nodes considered in the linear arithmetic analysis.

Conditional nodes have a predicate of the form $e = 0$, where e is a linear arithmetic expression. Non-deterministic conditionals, represented by $*$, denote that the control can flow to either branch irrespective of the program state before the conditional. They are used as a safe abstraction of conditional guards that our abstraction cannot handle precisely.

A join node has two incoming edges. Note that a join node with more than two incoming edges can be reduced to several join nodes each with two incoming edges.

Note that this is an intra-procedural analysis. The procedure calls have to be abstracted using non-deterministic assignments.

Notation

We use the following notation related to the above program model.

- $n_{\mathtt{a}}$: Number of (deterministic and non-deterministic) assignment nodes.
- $n_{\mathtt{c}}$: Number of (deterministic and non-deterministic) conditional nodes. Also, number of join nodes.
- n : Number of nodes. Note that $n = n_{\mathtt{a}} + 2n_{\mathtt{c}}$.
- $n_{\mathtt{ap}}$: Maximum number of assignment nodes in any procedure.

- $n_{\texttt{cp}}$: Maximum number of conditional nodes in any procedure.
- $k_{\texttt{v}}$: Maximum number of program variables visible at any program point.
- $k_{\texttt{u}}$: Maximum number of program variables visible at any point in a program loop (maximal strongly connected component) and also updated inside that loop.

A standard optimization for several algorithms that we describe in this dissertation involves converting the program into SSA form [CFR$^+$90]. With regard to that, we use the following notation:

- $n_{\texttt{s}}$: Number of total assignment statements (both phi assignments and non-phi assignments) in SSA version of the program.

It has been reported [CFR$^+$90] that the ratio of the number of new phi-assignments introduced (as a result of SSA conversion) to the number of original assignments typically varies between 0.3 to 2.8 (i.e., $1.3n_{\texttt{a}} \leq n_{\texttt{s}} \leq 3.8n_{\texttt{a}}$) irrespective of program size.

For describing a bound on the number of steps required for fixed-point computation, we use the following notation:

- β : Maximum number of back-edges in any program loop

For a structured flow-graph, β denotes the maximum loop nesting depth. Based on the experiments that we carried out, we noticed β to be a small constant in practice, usually bounded above by 3.

2.2.1 Basic Algorithm

The random interpreter computes a sample at each program point. The number of states in each sample can be chosen to be $t = 3k_{\texttt{v}}/2 + 3$. Each state in a sample maps program variables (which are visible at the corresponding program point) to values from the field $\mathbb{F}_p$. The prime p modulo which the random interpreter performs arithmetic is chosen randomly from a large enough set of primes. Experiments suggest that a randomly chosen 32 bit prime number is good enough. However, our conservative theoretical analysis implies that in the worst case, we need to work with larger prime numbers. For a detailed discussion, see Section 2.2.2.

For notational convenience, we extend the definition of a state to also include an undefined state, denoted by $\perp$. We say that $\perp \models e = 0$ for any expression e. Similarly, we also extend the definition of a sample to include an undefined sample, also denoted by $\perp$, that consists of all $\perp$ states.

The random interpreter computes the samples at different program points by performing a forward analysis on each procedure in the program. In presence of loops, the random interpreter goes around loops until a fixed point is reached. This issue is further discussed in Section 2.2.3. We now describe the action of the random interpreter on the flowchart nodes shown in Figure 2.6.

Initialization: At procedure entry, the random interpreter starts with an initial sample S^0 all of whose states are initialized to random values for all variables.

$$S_i^0(x) = \texttt{Rand}() \quad \text{for } 1 \leq i \leq t \text{ and all variables } x$$

The random interpreter initializes the samples at all other program points to $\perp$.

Assignment Node: See Figure 2.6 (a).
If the sample S' before the assignment node is undefined, then the sample S after the assignment node is also undefined.

$$S = \perp, \text{ if } S' = \perp.$$

Else, the random interpreter behaves almost like a concrete interpreter. For the assignment $x := e$, it transforms each state S'_i in the sample S' by setting x to the value of e in that state, which is denoted by $\texttt{Eval}(e, S'_i)$.

$$S_i = S'_i[x \leftarrow \texttt{Eval}(e, S'_i)]$$

The random interpreter evaluates an expression e in a state ρ using the usual interpretation of the arithmetic operations, but over field $\mathbb{F}_p$.

$$\begin{aligned}
\texttt{Eval}(c, \rho) &= c \bmod p \\
\texttt{Eval}(x, \rho) &= \rho(x) \\
\texttt{Eval}(e_1 \pm e_2, \rho) &= (\texttt{Eval}(e_1, \rho) \pm \texttt{Eval}(e_2, \rho)) \bmod p \\
\texttt{Eval}(c \times e, \rho) &= (c \times \texttt{Eval}(e, \rho)) \bmod p
\end{aligned}$$

Non-deterministic Assignment Node: See Figure 2.6 (b).
If the sample S' before the non-deterministic assignment node is undefined, then the sample S after the non-deterministic assignment node is also undefined.

$$S = \perp, \text{ if } S' = \perp.$$

Else, the random interpreter processes the assignment $x :=?$ by transforming each state S'_i in the sample S' by setting x to some fresh random value.

$$S_i = S'_i[x \leftarrow \texttt{Rand}()]$$

Conditional Node: See Figure 2.6 (c).
If the sample S before the conditional node is undefined, then the samples S^1 and S^2 on the two branches of the conditional node are also undefined.

$$S^1 = \perp \text{ and } S^2 = \perp, \text{ if } S = \perp$$

Else, if the random interpreter concludes that the conditional $e = 0$ is always true (or always false), then it executes only the true (or false) branch of the conditional.

$$S^1 = S \text{ and } S^2 = \perp, \text{ if } S \models e = 0$$
$$S^1 = \perp \text{ and } S^2 = S, \text{ if } S \models e - c = 0 \text{ for some non-zero constant } c$$

Otherwise, the random interpreter executes both branches of the conditional $e = 0$.

$$S^1 = \texttt{Adjust}(S, e) \text{ and } S^2 = S$$

Non-deterministic Conditional Node: See Figure 2.6 (d).
The random interpreter executes both branches of the non-deterministic conditional node and simply copies the sample before the conditional on its two branches.

$$S^1 = S \text{ and } S^2 = S$$

Join Node: See Figure 2.6 (e).
If any one of the samples before the join node is undefined, the random interpreter assigns the other sample before the join node to the sample after the join node.

$$S = \begin{cases} S^1 & \text{if } S^2 = \perp \\ S^2 & \text{if } S^1 = \perp \end{cases}$$

Otherwise the random interpreter chooses t weights independently and u.a.r. from $\mathbb{F}_p$ and performs the affine join of the two samples before the join node with respect to those randomly chosen weights.

$$S = \phi_{[w_1,\ldots,w_t]}(S^1, S^2), \text{ where } w_i = \texttt{Rand}()$$

Verifying and Discovering Linear Equalities

After fixed-point computation, the resulting samples can be used to verify whether a given linear equality holds at a given program point or not. Consider any program point π. Let S be the sample computed by the random interpreter at π after fixed-point computation. If $S = \perp$, then the random interpreter declares the program point π to be unreachable. Else, the random interpreter verifies a linear equality $e_1 = e_2$ at program point π if $\texttt{Eval}(e_1, S_i) = \texttt{Eval}(e_2, S_i)$ for all states S_i in the sample S.

For discovering linear equalities at program point π, the random interpreter extracts relationships from the sample S. This is done by assuming a relationship of the form $\alpha + \sum_j \alpha_j x_j = 0$ among the variables x_j's that are visible at π, and then solving for the unknowns α and α_j's from the following set of simultaneous linear equations:

$$\alpha + \sum_j \alpha_j S_i(x_j) = 0 \qquad 1 \le i \le t$$

The above system of equations may have a parametrized solution (instead of a unique solution). From the parametrized solution, we may obtain a linearly independent set of solutions by repeatedly plugging 1 for one of the parameters and 0 for the rest. A more useful set of linear equalities may be those that involve few variables (as opposed to potentially all visible variables), e.g., variable equalities, constant variables, or induction variables. These may be discovered by assuming more specific templates like $x = \alpha$ or $x = \alpha_1 y + \alpha_2$, and solving for the unknowns α's as above.

Note that the coefficients of the linear equalities thus discovered are expressed modulo p (since the random interpreter performs arithmetic modulo p to avoid the problem of dealing with large numbers). The challenge now is to obtain the original rational coefficients, assuming that these rational coefficients involve integers with small absolute value. In other words, given $z \in \mathbb{F}_p$, we want to obtain small integers m_1 and m_2 such that $\frac{m_1}{m_2} = z \bmod p$, i.e., $m_1 = m_2 z \bmod p$. Note that if both m_1 and m_2 have absolute value less than $\sqrt{\frac{p}{2}}$, then m_1 and m_2 are unique.

```
Rationalize($z$,$p$)
    $u_1 = (z, 1)$;
    $u_2 = (p, 0)$;
    do
        if ($\|u_1\|$ > $\|u_2\|$) then swap $u_1$ and $u_2$;
        $a := \left\lfloor \frac{\langle u_1, u_2 \rangle}{\|u_1\|^2} \right\rceil$;
        $u_2 := u_2 - a u_1$;
    while ($\|u_1\| > \|u_2\|$);
    return $u_1$;
```

Figure 2.7: A procedure to obtain small rational coefficients from their images in $\mathbb{F}_p$. Here $\langle u_1, u_2 \rangle$ denotes the inner product of vectors u_1 and u_2, i.e., if $u_1 = (a_1, b_1)$ and $u_2 = (a_2, b_2)$, then $\langle u_1, u_2 \rangle = a_1 a_2 + b_1 b_2$. $\|u\|$ denotes the Euclidean norm $\sqrt{\langle u, u \rangle}$, and $\lfloor f \rceil$ denotes the integer closest to f.

Given $z \in \mathbb{F}_p$, the problem of finding smallest m_1 and m_2 such that $\frac{m_1}{m_2} = z \mod p$ is equivalent to the problem of finding the shortest (in terms of the Euclidean norm) non-zero vector in the set $A = \{(m_1, m_2) \parallel m_1 - m_2 z = 0 \bmod p\}$. The latter problem can be solved by using *lattice reduction* techniques [Car02]. Since the set A is 2-dimensional in this case, we can simply use *Gaussian reduction* algorithm [Gau86], which is similar to the Euclidean algorithm for computing the greatest common divisor of two numbers. The procedure `Rationalize` shown in Figure 2.7 implements this algorithm and returns the pair m_1 and m_2, given z and p.

The correctness of the procedure `Rationalize` follows from the invariant that both vectors u_1 and u_2 belong to the set A and their norm decreases in each iteration of the while loop. The while loop terminates with u_1 being the shortest vector in set A and u_2 being the next shortest vector. We now sketch a proof that the while loop in the procedure `Rationalize` terminates in at most $\lceil \log 2p \rceil$ iterations. The value of $\|u_2\|$ in the first iteration is p. Its value in the last iteration is at least $\sqrt{\frac{p}{2}}$, which is a lower bound for the length of the second shortest vector in the set A. (This is because there can be at most one pair of m_1 and m_2 such that $\frac{m_1}{m_2} = z \mod p$ and $|m_1|, |m_2| < \sqrt{\frac{p}{2}}$. Hence, it must be the case that at least one element of either the shortest vector or the second shortest vector must have absolute value at least $\sqrt{\frac{p}{2}}$. This implies that the length of the second shortest vector is bounded below by $\sqrt{\frac{p}{2}}$.) It can be proved that $\|u_2\|$ decreases by a factor of at least $\sqrt{2}$ in each iteration. Hence, the loop terminates in at most $\lceil \log_{\sqrt{2}} \left(\frac{p}{\sqrt{\frac{p}{2}}} \right) \rceil$ iterations. However, experiments show that on the average 6 iterations are needed for a 32-bit prime.

We now describe some heuristics (as an alternative to implementing the procedure `Rationalize`) to discover the original rational coefficients from their images in $\mathbb{F}_p$. We can compute and store the following mapping I_p (indexed by its image) for a randomly chosen prime p and some small integer constant c beforehand.

$$I_p\left(\frac{m_1}{m_2}\right) = m_1 \times m_2^{-1} \bmod p \qquad -c \leq m_1 \leq c,\ 1 \leq m_2 \leq c$$

Hence, given z, we can lookup the store to immediately output m_1 and m_2 such that $\frac{m_1}{m_2} = z \bmod p$. This approach works if the absolute values of the numerator and denominator are at most c.

Another alternative is to assume that the denominators of the coefficients of the linear equalities are 1, or in fact any known constant m. In this case, given z and m, we can estimate m_1 such that $z = \frac{m_1}{m} \bmod p$ as follows:

$$m_1 = \begin{cases} m_1' & \text{if } m_1' < \frac{p}{2} \\ m_1' - p & \text{otherwise} \end{cases}$$

$$m_1' = (z \times m) \bmod p$$

In either of the above solutions (for obtaining rational numbers from their images in $\mathbb{F}_p$), we need to verify the linear equalities thus discovered. This is because we do not know beforehand whether the assumption on the numerators and denominators (of the rational coefficients in the linear equalities) being small hold or not. Hence, we need to run the random interpreter again with a new randomly chosen prime p' and verify the linear equalities discovered in the first round.

Optimization

Maintaining a sample explicitly at each program point is expensive (in terms of time and space complexity) and redundant. For example, consider the samples before and after an assignment node $x := e$. They differ (at most) only in the values of variable x.

An efficient way to represent samples at each program point is to convert the program into minimal SSA form [CFR$^+$90] and then maintain one global sample for the program instead of maintaining a sample at each program point. The values of a variable x in the sample at program point π are represented by the values of the variable x_π in the global sample, where x_π denotes the renamed version of variable x at program point π after the SSA conversion. To account for the fact that the Adjust operation updates the sample on the true side of a conditional, when converting a program into SSA form, we assume that the true branch of a conditional node updates the values of all original program variables that were visible immediately before the conditional node. Under such a representation of one global program sample, interpreting an assignment node simply involves updating the values of the modified variable in the global sample. Interpreting a join node involves updating the values of ϕ variables at that join point in the global sample.

2.2.2 Error Probability Analysis

In this section, we estimate the probability that the random interpreter gives a false judgment, i.e., it verifies or discovers an incorrect linear equality. Theorem 2, Theorem 3, and Theorem 4 in this section together state the total error probability of the random interpreter. The analysis is complicated because of the Adjust operations performed by the random interpreter (to reason about equality predicates in conditionals). A simpler analysis is possible when the random interpreter does not perform any Adjust operation, which means that all conditionals are treated as non-deterministic. A busy reader may skip this section and refer to the section on page 42 for the idea behind this simpler proof.

For the purpose of the random interpreter's analysis, we introduce an abstract interpreter that computes, at each program point, a sound approximation of the set of linear equalities that hold at that program point. The abstract interpreter represents the set of linear equalities U at each program point by a set of states T that span the subspace defined by the linear equalities U. We also estimate the maximum number of bits b required to represent the numerator and denominator of the values of variables in the states in T.

We write $U \Rightarrow e_1 = e_2$ to denote that the conjunction of linear equalities in U imply $e_1 = e_2$. We use the notation $U \Rightarrow_p e_1 = e_2$ to denote that the conjunction of linear equalities in U imply $e_1 = e_2$ over the prime field $\mathbb{F}_p$. We write $U^1 \cap U^2$ for the set of linear equalities that are implied by both U^1 and U^2. (This operation is sometimes called the union of affine spaces [Kar76]). Finally, we write $U[e/x]$ for the linear equalities that are obtained from those in U by substituting e for x. For notational convenience, we let

$\perp$ represent an inconsistent set of linear equalities. We also say that $\perp \Rightarrow e_1 = e_2$ and $\perp \Rightarrow_p e_1 = e_2$ for any linear equality $e_1 = e_2$. Let s be a bound on the size of all expressions e that occur in the assignment node $x := e$ or conditional node $e = 0$ in terms of the number of additions. Let c_m be a bound on the absolute value of the coefficients that occur in these expressions. With these definitions we now define the action of the abstract interpreter over the flowchart nodes shown in Figure 2.6.

Initialization: The abstract interpreter initializes $U = \perp$, $T = \{\}$ and $b = -1$ at all points other than the procedure entry. At procedure entry, the abstract interpreter starts with the following initial configuration. Let τ_x denote the state that maps input variable x to 1 and all other input variables to 0. Let $\tau_{\texttt{all}}$ denote the state that maps all input variables to 1.

$$\begin{aligned} U &= \emptyset \\ T &= \{\tau_x \parallel x \text{ is an input variable }\} \cup \{\tau_{\texttt{all}}\} \\ b &= 1 \end{aligned}$$

Assignment Node: See Figure 2.6 (a).
If $U' = \perp$, then $U = \perp$, $T = \emptyset$, and $b = -1$. Else,

$$\begin{aligned} U &= \{x = e[x'/x]\} \cup U'[x'/x], \text{ where } x' \text{ is a fresh variable} \\ T &= \{\tau[\texttt{Eval}(e,\tau)/x] \parallel \tau \in T'\} \\ b &= \log s + s(\log c_m + b') \end{aligned}$$

Non-deterministic Assignment Node: See Figure 2.6 (b).
If $U' = \perp$, then $U = \perp$, $T = \emptyset$, and $b = -1$. Else,

$$\begin{aligned} U &= U'[x'/x], \text{ where } x' \text{ is a fresh variable} \\ T &= T' - \{\tau\} \cup \{\tau[0/x], \tau[1/x]\}, \text{ where } \tau \text{ is any state from } T' \\ b &= b' \end{aligned}$$

The fact that T spans the subspace defined by U (assuming that T' spans the subspace defined by U') follows from the following observation. Let e be any expression that involves x with non-zero coefficient. Then, for any state τ, at least one of $\texttt{Eval}(e, \tau[0/x])$ or $\texttt{Eval}(e, \tau[1/x])$ is non-zero.

Conditional Node: See Figure 2.6 (c).
If $U \Rightarrow e = 0$, then $U^1 = U$, $T^1 = T$, $b^1 = b$, $U^2 = \bot$, $T^2 = \emptyset$, and $b^2 = -1$. Else, if $U \Rightarrow e - c = 0$ for some non-zero constant c, then $U^1 = \bot$, $T^1 = \emptyset$, $b^1 = -1$, $U^2 = U$, $T^2 = T$, and $b^2 = b$. Else,

$$\begin{aligned}
U^1 &= U \cup \{e = 0\} \\
T^1 &= \texttt{Adjust2}(T, e) \\
b^1 &= 3\log s + 3s\log c_m + b(3s+1) + 2 \\
U^2 &= U \\
T^2 &= T \\
b^2 &= b
\end{aligned}$$

The `Adjust2` function involves generating new points τ by selecting any two points τ_1 and τ_2 from T that are not equidistant from the hyperplane $e = 0$ and drawing a line between them to intersect $e = 0$ at τ.

$$\texttt{Adjust2}(T, e) = \{\phi_w(\tau_1, \tau_2) \parallel \tau_1, \tau_2 \in T, \texttt{Eval}(e, \tau_1) \neq \texttt{Eval}(e, \tau_2)\}$$
$$\text{where } w \equiv \frac{\texttt{Eval}(e, \tau_2)}{\texttt{Eval}(e, \tau_2) - \texttt{Eval}(e, \tau_1)}$$

The fact that T^1 spans the subspace defined by U^1 (assuming that T spans the subspace defined by U) follows from the following observation. If all states in $\texttt{Adjust2}(e, T)$ satisfy $g = 0$, then all states in T satisfy $g + \alpha e = 0$ for some constant α. (In fact, $\alpha = \frac{-\texttt{Eval}(g,\tau)}{\texttt{Eval}(e,\tau)}$ where τ is some state from T.) Also, if all states in T satisfy $g = 0$, then all states in $\texttt{Adjust2}(e, T)$ satisfy $g = 0$ since all states in $\texttt{Adjust2}(e, T)$ are affine combinations of the states in T.

Non-deterministic Conditional Node: See Figure 2.6 (d).
$U^1 = U^2 = U$, $T^1 = T^2 = T$, and $b^1 = b^2 = b$.

Join Node: See Figure 2.6 (e).
If $U^1 = \bot$, then $U = U^2$, $T = T^2$, and $b = b^2$. Else if $U^2 = \bot$, then $U = U^1$, $T = T^1$, and

$b = b^1$. Else,

$$U = U^1 \cap U^2$$
$$T = T^1 \cup T^2$$
$$b = max(b^1, b^2)$$

The abstract interpreter defined above is less efficient than the random interpreter (otherwise, there would be no need for the random interpreter). We use this abstract interpreter only to state and prove the soundness and completeness results of the random interpreter.

Given a program, the sets of linear equalities U computed by the abstract interpreter at every point in the program (after fixed-point computation) are uniquely defined. Corresponding to each such U, there is a random sample S (which depends on the random choices made by the random interpreter). We now prove that any sample S satisfies exactly the linear equalities that are implied by the corresponding U with high probability over the random choices made by the random interpreter.

The following theorem states that any sample S computed by the random interpreter satisfies all the linear equalities that are implied by the corresponding set of linear equalities U computed by the abstract interpreter. Note that this statement does not involve any error probability.

Theorem 1 [Completeness Theorem] *Let S be a sample computed by the random interpreter at some program point. Let U be the corresponding set of linear equalities computed by the abstract interpreter at that program point. Let $g = 0$ be a linear equality such that $U \Rightarrow g = 0$. Then, $S \models g = 0$.*

The proof of Theorem 1 is by induction on the number of samples computed by the random interpreter, and is based on Lemma 1 and Lemma 3. The proof of Theorem 1 is given in Appendix A.1.

We now show that there is a very small probability that any sample S computed by the random interpreter satisfies any linear equality that is not implied by the corresponding set of linear equalities U computed by the abstract interpreter. We estimate this error probability of the random interpreter in two steps. First, we estimate the error probability of the random interpreter assuming that the linear equalities are to be discovered over the prime field $\mathbb{F}_p$ (Theorem 2). Then, we estimate the additional error probability that results from performing the computations over the prime field $\mathbb{F}_p$ instead of the infinite field of rationals (Theorem 3 and Theorem 4).

Step 1:

We first introduce a useful definition.

Definition 1 [Sound Sample] Let S be a sample computed by the random interpreter at some program point. Let U be the corresponding set of linear equalities computed by the abstract interpreter at that program point. We say that sample S is *sound* if for all linear equalities $g = 0$ such that $U \not\Rightarrow_p g = 0$, we have that $S \not\models g = 0$.

Let $\mathcal{A}_S$ denote the event that all samples computed by the random interpreter before computing sample S are sound. The notation $\Pr(\mathcal{E}_1 \parallel \mathcal{E}_2)$ denotes the probability of event $\mathcal{E}_1$ (over the corresponding random choices) given the occurrence of event $\mathcal{E}_2$.

Lemma 5 [Soundness Lemma] *Let S be a sample computed by the random interpreter at some program point π. Let q be the maximum number of adjust and join operations performed by the random interpreter on any path (leading to π from procedure entry) before computing sample S. Let U be the corresponding set of linear equalities computed by the abstract interpreter at that program point. Let $g = 0$ be a linear equality such that $U \not\Rightarrow g = 0$. Let $\tilde{S}$ be some subset of ℓ states from the sample S. If $t \geq 3k_{\mathrm{v}}/2 + 3$ and $p \geq (q+2)(2(q+1)^2 + t + 2)$, then* $\Pr(\tilde{S} \models g = 0 \parallel \mathcal{A}_S) \leq \left(\frac{q+1}{p}\right)^{\ell}$.

The proof of Lemma 5 is by induction on the number of samples computed by the random interpreter, and is based on Lemma 2 and Lemma 4. The proof of Lemma 5 is given in Appendix A.2.

There are two things worth mentioning about the statement of the soundness theorem. Note that the soundness theorem implies a bound on $\Pr(S \models g = 0 \parallel \mathcal{A}_S)$. However, in order for the inductive proof to work, the soundness theorem actually states a bound for $\Pr(\tilde{S} \models g = 0 \parallel \mathcal{A}_S)$, where $\tilde{S}$ is any subset of S. Also, note that the soundness theorem provides a bound for $\Pr(S \models g = 0 \parallel \mathcal{A}_S)$ rather than $\Pr(S \models g = 0)$. Again, this is needed for the inductive proof to work. As we will see later, a bound on $\Pr(S \models g = 0 \parallel \mathcal{A}_S)$ is sufficient to bound the total error probability of the random interpreter.

According to Lemma 5, given any relationship not verified by the abstract interpreter, the probability (over the random choices made by the random interpreter) that the relationship is verified by the random interpreter is extremely small (assuming that some samples computed by the random interpreter are sound, which will be the case with very high probability).

It follows from the discussion after Theorem 5 in the next section that the random interpreter goes around each loop at most $(k_{\mathtt{u}}+1)\beta$ times for fixed-point computation. We use this observation along with Lemma 5 to prove Theorem 2, which establishes an upper bound on the probability that the random interpreter is unsound.

Theorem 2 [Probabilistic Soundness Theorem] *Let $q = 2n_{\mathtt{cp}}(k_{\mathtt{u}}+1)\beta$ and $q' = n(k_{\mathtt{u}}+1)\beta$. The probability that all random samples computed by the random interpreter satisfy only those linear equalities that are implied by the corresponding set of linear equalities computed by the abstract interpreter (over the prime field $\mathbb{F}_p$) is at least $1-q'\left(\frac{1}{p-1}\right)^{2t/3-k_{\mathtt{v}}}$, if $t \geq 3k_{\mathtt{v}}/2+3$ and $p \geq (q+2)(2(q+1)^2+t+2)$.*

Proof. Let $S^1, \ldots, S^m$ be the different samples computed by the random interpreter in that order. Let $U^1, \ldots, U^m$ be the corresponding sets of linear equalities computed by the abstract interpreter. Let $\mathcal{C}_i$ be the event that sample S^i is not sound, i.e., there exists some linear equality $g=0$ such that $S^i \models g=0$ and $U^i \not\Rightarrow g=0$.

$$\begin{aligned}
\Pr(\mathcal{C}_1 \vee \ldots \vee \mathcal{C}_m) &= \sum_{i=1}^{m} \Pr(\mathcal{C}_i \wedge \bigwedge_{j=1}^{i-1} \neg\mathcal{C}_j) \\
&= \sum_{i=1}^{m} \Pr(\mathcal{C}_i \wedge \mathcal{A}_{S^i}) \\
&\leq \sum_{i=1}^{m} \Pr(\mathcal{C}_i \parallel \mathcal{A}_{S^i}) \\
&\leq \sum_{i=1}^{m} \sum_{g,\ \text{s.t.}\ U^i \not\Rightarrow g=0} \Pr(S^i \models g=0 \parallel \mathcal{A}_{S^i}) \\
&\leq \sum_{i=1}^{m} \sum_{g,\ \text{s.t.}\ U^i \not\Rightarrow g=0} \left(\frac{q+1}{p}\right)^t \quad \text{(follows from Lemma 5)}
\end{aligned}$$

Note that there are $\frac{p^{k_{\mathtt{v}}+1}-p}{p-1}$ different linear equalities with coefficients from $\mathbb{F}_p$ between $k_{\mathtt{v}}$ variables and hence this is an upper bound on the number of linear equalities not implied by U. Also, note that the random interpreter goes around each loop at most

$(k_{\mathtt{u}}+1)\beta$ times (follows from the discussion after Theorem 5). Hence, $m \leq n(k_{\mathtt{u}}+1)\beta = q'$. Thus,

$$\begin{aligned}\Pr(\mathcal{C}_1 \vee \ldots \vee \mathcal{C}_m) &\leq q' \times \frac{p^{k_{\mathtt{v}}+1}-p}{p-1} \times \left(\frac{q+1}{p}\right)^t \\ &\leq q' \times \left(\frac{1}{p-1}\right)^{2t/3-k_{\mathtt{v}}}\end{aligned}$$

□

Step 2:

In this section, we estimate the additional error probability that results from reducing the problem of finding linear equalities in a program with variables that take rational values to one in which the program variables take values in some random prime field $\mathbb{F}_p$. We also estimate the error probability in the process (described in Section 2.2.1) of inferring the rational coefficients of the linear equalities from their values in the prime field. This error probability is a function of the size of the set from which the prime is chosen randomly; hence, it suggests how big the size of this set should be in order to obtain a specific upper bound on the error probability.

Performing computations over a prime field preserves all true linear equalities, but may introduce some spurious linear equalities. For example, consider the following program fragment, where c is some prime number.

```
if (*) then $x := 1$ else $x := c+1$;
assert ($x = 1$);
```

The assertion at the end of the program is false, but if the arithmetic is performed over the prime field $\mathbb{F}_p$ for $p = c$, then the assertion becomes true. However, note that the probability of choosing the prime number p to be c is small since the prime number p is chosen from a large enough set of primes. It follows from Theorem 3 (stated and proved below) that, in general, the probability of such spurious linear equalities being introduced is small.

The process of discovering the true coefficients of linear equalities from the coefficients expressed in the prime field can also introduce some error. For example, consider the following program fragment:

```
$x := c$;
```

The linear equality $x = c$ holds at the end of the above program fragment. Suppose $p \leq c \leq 3p/2$. Let $c' = c \bmod p$. Then, the technique suggested in Section 2.2.1 will yield the incorrect linear equality $x = c'$, if $c' \bmod p' = c \bmod p'$, where p' is the second randomly chosen prime for performing random interpretation over the prime field $\mathbb{F}'_p$ to verify the equalities discovered during the first round. However, it follows from Property 1 (stated below) that the probability of choosing the prime number p' to be such that $c' \bmod p' = c \bmod p'$ is small since the prime number p' is chosen from a large enough set of primes. It follows from Theorem 4 (stated and proved below) that in general the probability of discovering such incorrect coefficients in linear equalities is small.

Before stating and proving the theorems that bound the desired error probabilities, we first state a useful property.

Property 1 *The probability that two distinct b bit integers are equal modulo a randomly chosen prime from $[1, p_m]$ is at most $\frac{1}{u}$ for $p_m \geq 2ub \log(ub)$.*

The proof of Property 1 follows from the prime number theorem which states that the number of prime numbers less than x is at least $\frac{x}{\log x}$.

Refer to the description of the abstract interpreter in the earlier part of this section. Observe that there are at most $n_{\texttt{ap}}(k_{\texttt{u}}+1)$ evaluations of an assignment node, and $n_{\texttt{cp}}(k_{\texttt{u}}+1)$ evaluations of a conditional node for any procedure. Hence, the number of bits b_m required to represent the numerator and denominator in any state in T computed by the abstract interpreter can be bounded as follows:

$$b_m \leq (3\log s + 3s\log c_m + 2)(3s+2)^{n_{\texttt{cp}}(k_{\texttt{u}}+1)}(s+1)^{n_{\texttt{ap}}(k_{\texttt{u}}+1)}$$

We now state and prove the theorems that bound the desired error probabilities.

Theorem 3 *The probability that performing arithmetic over the prime field $\mathbb{F}_p$ when p is chosen randomly from $[1, p_m]$, introduces any spurious linear equalities is bounded above by $\frac{n}{2^a}$, if $p_m = 2^{a+1}b'_m \log(2^a b'_m)$, where $b'_m = (b_m+1)(k_{\texttt{v}}+1)^2$.*

Proof. For any program point π, let $\mathcal{E}_\pi$ be the event that the number of (linearly independent) linear equalities computed at π (after fixed-point computation) remains the same if the computations are performed over some (randomly chosen) prime field $\mathbb{F}_p$. Let T^π be the set of states computed by the abstract interpreter (after fixed-point computation) at program point π. Let k be the number of linearly independent linear equalities implied by

T^π. Let $k' = k_{\texttt{v}} + 1 - k$. There exists a matrix of size $k' \times k'$ with non-zero determinant D_π whose columns correspond to some k' states from T^π and rows correspond to the values of some k' variables in those states. Note that $\Pr(\mathcal{E}_\pi) = \Pr(D_\pi \neq 0 \bmod p)$. The number of bits in the numerator and denominator of D_π is bounded above by $b'_m = (b_m + 1)(k_{\texttt{v}} + 1)^2$. Hence, it follows from Property 1 that $\Pr(\mathcal{E}_\pi) \leq \frac{1}{2^a}$, if we choose the prime number p randomly from $[1, p_m]$, where $p_m = 2^{a+1} b'_m \log(2^a b'_m)$. The desired result now follows from the union bound over the probabilities of the events $\mathcal{E}_\pi$ for all n program points π. □

Theorem 4 *Suppose the prime p' used for verifying the linear equalities discovered in the first round is chosen randomly from $[1, p_m]$. Let c'_m be the bound on the absolute value of the coefficients of the linear equalities discovered in the first round. Then, the probability of incorrect verification of the coefficients is bounded above by $\frac{nk_{\texttt{v}}}{2^a}$, if $p_m = 2^{a+1} b'_m \log(2^a b'_m)$, where $b'_m = (b_m + c'_m)(k_{\texttt{v}} + 1) + \log(k_{\texttt{v}} + 1)$.*

Proof. For any program point π and any linear equality $e = 0$ that does not hold at π, let $\mathcal{F}_{\pi,e}$ be the event that $e = 0$ does not hold at π if the computations are performed over some (randomly chosen) prime field $\mathbb{F}_{p'}$. Let T^π be the set of states computed by the abstract interpreter (after fixed-point computation) at program point π. There must be some state τ in T^π such that $\texttt{Eval}(e, \tau) \neq 0$. Note that $\Pr(\mathcal{F}_{\pi,e}) = \Pr(\texttt{Eval}(e, \tau) \neq 0 \bmod p')$. The number of bits in the numerator and denominator of $\texttt{Eval}(e, \tau)$ is bounded above by $b'_m = (b_m + c'_m)(k_{\texttt{v}} + 1) + \log(k_{\texttt{v}} + 1)$. Hence, it follows from Property 1 that $\Pr(\mathcal{F}_{\pi,e}) \leq \frac{1}{2^a}$, if we choose the prime number p' randomly from $[1, p_m]$, where $p_m = 2^{a+1} b'_m \log(2^a b'_m)$. The desired result now follows from the union bound over the probabilities of the events $\mathcal{F}_{\pi,e}$ for all n program points π and at most $k_{\texttt{v}}$ linearly independent linear equalities $e = 0$ at those points. □

Theorem 3 and Theorem 4 suggest that the random interpreter must perform arithmetic with primes that require $O(n_{\texttt{ap}} k_{\texttt{u}})$ bits for representation (assuming $n_{\texttt{cp}} = O(n_{\texttt{ap}})$ and s and c_m to be constants). However, we feel that this is a conservative analysis, and we do not know of any program that illustrates this worst-case behavior. Experiments show that 32-bit primes are good enough.

2.2.3 Fixed-point Computation

In presence of loops (maximal strongly connected components) in procedures, the random interpreter goes around loops (just like any abstract interpreter or a data-flow analyzer) until a fixed point is reached. We say that the random interpreter reaches a fixed point across a loop when the sets of linear equalities (computed by the abstract interpreter introduced in the previous section) corresponding to the samples computed by the random interpreter reach a fixed point. Note that the samples computed by the random interpreter themselves do not reach a fixed point, but the linear equalities represented by them do.

The elements of the abstract lattice over which the abstract interpreter performs computations are sets of linear equalities between program variables. These elements are ordered by the implication relationship (i.e., if U^2 is above U^1 in the abstract lattice, then $U^1 \Rightarrow U^2$). The following theorem provides a bound on the number of steps required to reach a fixed point across a loop.

Theorem 5 [Fixed Point Theorem] *Let $U^1, \ldots, U^m$ be the sets of linear equalities that are computed by the abstract interpreter at some point π inside a loop in successive iterations of that loop such that $U^i \not\equiv U^{i+1}$. Then, $m \leq k_{\mathbf{u}} + 1$, where $k_{\mathbf{u}}$ is the number of variables that are visible at π as well as updated inside that loop.*

Proof. Let V be the set of variables that are visible at π as well as updated inside the loop. For all $1 \leq i \leq m-1$, there exists a linear equality $g_i = 0$ such that $U^i \Rightarrow g_i = 0$ and $U^{i+1} \not\Rightarrow g_i = 0$. These g_i's involve non-trivial occurrences of some variable from V. Also, these g_i's are linearly independent of each other. Suppose for the purpose of contradiction that $m > k_{\mathbf{u}} + 1$. Then, we can solve for the variables in V using $g_2 = 0, \ldots, g_{k_{\mathbf{u}}+1} = 0$ and obtain g_1' from g_1 using these substitutions such that g_1' does not involve any variable from V. Since $U^1 \Rightarrow g_i = 0$ and $U^2 \Rightarrow g_i = 0$ for all $2 \leq i \leq m-1$, it follows that $U^1 \Rightarrow g_1' = 0$ and $U^2 \not\Rightarrow g_1' = 0$. This is a contradiction since g_1' does not involve any variable from V. $\square$

One way to detect when the random interpreter has reached a fixed point is to compare the rank of the samples (viewed as matrices) at relevant locations in two successive iterations of a loop. The rank of a sample is one more than the number of variables minus the number of linearly independent linear equalities that the sample satisfies. Thus, if the rank has stabilized, the number of linearly independent linear equalities satisfied by S has been stabilized, and so has the set of linear equalities satisfied by S. Computing rank of a matrix of size $t \times k_{\mathtt{v}}$ takes time $O(tk_{\mathtt{v}}^2)$, and hence it may be an expensive operation. A better strategy may be to iterate around a loop $(k_{\mathtt{u}}+1)\beta$ times, where β is the number of back-edges in the loop. Note that it is guaranteed that a fixed point will be reached after iterating across a loop $(k_{\mathtt{u}}+1)\beta$ times. This is because if a fixed point is not reached, then the set of linear equalities corresponding to at least one of the samples at the target of the back-edges must change, and it follows from Theorem 5 that there can be at most $k_{\mathtt{u}}$ such changes at each program point.

2.2.4 Computational Complexity

The cost of each adjust and affine join operation performed by the random interpreter is $O(k_{\mathtt{v}}t)$, assuming unit cost for each arithmetic operation. Each assignment operation takes $O(t)$ time (assuming we perform the optimization discussed in Section 2.2.1). The random interpreter performs a maximum of $2(k_{\mathtt{u}}+1)\beta n_{\mathtt{c}}$ adjust and affine join operations and $(k_{\mathtt{u}}+1)\beta n_{\mathtt{a}}$ assignment operations. Since β is usually a small constant, the running time of the random interpreter is $O(n_{\mathtt{c}}k_{\mathtt{u}}k_{\mathtt{v}}t + n_{\mathtt{a}}k_{\mathtt{u}}t)$. Theorem 2 requires choosing t to be $3k_{\mathtt{v}}/2+3$ for probabilistic soundness; this yields an overall complexity of $O(n_{\mathtt{c}}k_{\mathtt{u}}k_{\mathtt{v}}^2+n_{\mathtt{a}}k_{\mathtt{u}}k_{\mathtt{v}})$ for the random interpreter.

The worst-case estimation in the previous section suggests that the arithmetic may need to be performed over a prime field that require $O(n_{\mathtt{ap}}k_{\mathtt{u}})$ bits for representation. However, we feel that this is a conservative analysis, and experiments suggest that 32-bit primes are good enough.

2.3 Special Case of Non-deterministic Conditionals

In this section, we consider the special case when all conditionals are abstracted as non-deterministic branches, meaning that the conditional can evaluate to either true or false irrespective of the program state before it. The random interpreter can reason about all linear equalities in this case. (We prove a more general result in Chapter 4 regarding the completeness of the random interpreter for inter-procedural reasoning when all conditionals are treated as non-deterministic.) The random interpreter does not perform any Adjust operation in presence of only non-deterministic conditionals, and this allows for proving better bounds on the computational complexity of the random interpreter.

First note that a stronger version of Lemma 5 holds:

Lemma 6 [Soundness Lemma] *Let S be a sample computed by the random interpreter at some program point π. Let q be the maximum number of join operations performed by the random interpreter on any path (leading to π from procedure entry) before computing sample S. Let U be the corresponding set of linear equalities computed by the abstract interpreter at that program point. Let $g = 0$ be a linear equality such that $U \not\Rightarrow_p g = 0$. Let $\tilde{S}$ be some subset of ℓ states from the sample S. Then,* $\Pr(\tilde{S} \models g = 0) \leq \left(\frac{q+1}{p}\right)^{\ell}$.

The proof of Lemma 6 is same as the proof for Lemma 5. Since no Adjust operation is performed, there is no requirement on t and p (unlike Lemma 5). Also the event $\mathcal{A}_S$ in Lemma 5 trivially reduces to true. Using Lemma 6, we can prove the following theorem.

Theorem 6 [Probabilistic Soundness Theorem] *Let $q = n_{\mathtt{cp}}(k_{\mathtt{u}}+1)\beta$ and $q' = n(k_{\mathtt{u}}+1)\beta$. The probability that all random samples computed by the random interpreter satisfy only those linear equalities involving k variables that are implied by the corresponding set of linear equalities computed by the abstract interpreter (over the prime field $\mathbb{F}_p$) is at least $1 - q'\left(\frac{1}{p-1}\right)^{2t/3-k}$, if $p \geq (q+1)^3 + 1$.*

Proof. The proof of Theorem 6 is similar to the proof of Theorem 2. Note that there are $\frac{p^{k+1}-p}{p-1}$ different linear equalities with coefficients from $\mathbb{F}_p$ between k variables and hence this is an upper bound on the number of linear equalities not implied by U. □

We can also prove better bounds on the size of the set from which the prime p needs to be chosen to perform the arithmetic. First, note that in the description of the abstract interpreter (introduced in Section 2.2.2), we can prove a better bound on b for the assignment node:

$$b = \log s + \log c_m + b'$$

Hence, the maximum number of bits b_m required to represent the numerator and denominator in each state τ computed by the abstract interpreter is bounded above as follows.

$$b_m \leq 1 + n_{\mathtt{ap}}(k_{\mathtt{u}} + 1)(\log s + \log c_m)$$

Plugging the above bound on b_m in Theorem 3 and Theorem 4 yields the error probability as a function of the size of the set from which the prime is chosen. Note that in this case Theorem 3 and Theorem 4 imply that the randomly chosen prime field over which the random interpreter performs arithmetic computations requires $O(\log n_{\mathtt{ap}})$ bits for representation of its elements.

Computational Complexity

We assume that the random interpreter performs the optimization of maintaining one global state in the SSA version of the program as discussed in Section 2.2.1. Under that optimization, a join operation reduces to processing phi-assignments at that join point. Assuming unit cost for each arithmetic operation, the cost of processing each assignment, both phi and non-phi, is $O(t)$. Fixed-point computation requires the random interpreter to go around each loop at most $(k_{\mathtt{u}} + 1)\beta$ times (as discussed after Theorem 5). Since β is usually a small constant, the running time of the random interpreter is $O(n_{\mathtt{s}} k_{\mathtt{u}} t)$. If the goal is to verify linear equalities, then it follows from Lemma 6 that we need to choose t to be a small constant (even $t = 1$ is sufficient) for probabilistic soundness. If the goal is to discover linear equalities involving k variables, then it follows from Theorem 6 that we need to choose t to be slightly greater than $3k/2$ (even $t = 3k/2 + 1$ is sufficient) for probabilistic soundness.

The worst-case estimation for the size of the set from which the prime p must be chosen randomly suggests that the arithmetic may need to be performed over a prime field whose elements require $O(\log n_{\mathtt{ap}})$ bits for representation. However, experiments show that 32-bit primes are good enough.

An Alternative Soundness Proof Strategy

We now show a new and simpler proof strategy for probabilistic soundness that also gives insight into how this algorithm could be extended beyond linear arithmetic. The probabilistic soundness proof given earlier is complicated to accommodate the Adjust operation performed by the random interpreter. If we ignore this operation, we can give a simpler proof of soundness in terms of polynomials. A straight-line sequence of assignments, involving only linear arithmetic, computes the values of variables at the end as linear polynomials in terms of the variables live on input. The overall effect of the affine join operation is to compute the weighted sum of these polynomials corresponding to each path. These weights themselves are non-linear polynomials in terms of the random weights w_i. For example, the values of a, b, x and y at the end of the program shown in Figure 2.2 (on page 13) can be written as follows:

$$\begin{aligned}
a &= w_1 \times 0 + (1 - w_1) \times (i - 2) \\
&= w_1(2 - i) + (i - 2) \\
b &= w_1 \times i + (1 - w_1) \times 2 \\
&= w_1(i - 2) + 2 \\
x &= w_2 \times (b - a) + (1 - w_2) \times (2a + b) \\
&= w_2 \times (2w_1(i - 2) + 4 - i) + (1 - w_2) \times (w_1(2 - i) + 2i - 2) \\
&= w_1w_2(3i - 6) + w_2(6 - 3i) + w_1(2 - i) + 2i - 2 \\
y &= w_2 \times (i - 2b) + (1 - w_2) \times (b - 2i) \\
&= w_2 \times (2w_1(2 - i) + i - 4) + (1 - w_2) \times (w_1(i - 2) + 2 - 2i) \\
&= w_1w_2(6 - 3i) + w_2(3i - 6) + w_1(i - 2) + 2 - 2i
\end{aligned}$$

Correspondingly, the two assertions at the end of the program can be written, respectively, as $(w_1w_2(3i-6)+w_2(6-3i)+w_1(2-i)+2i-2)+(w_1w_2(6-3i)+w_2(3i-6)+w_1(i-2)+2-2i) = 0$ and $w_1w_2(3i-6)+w_2(6-3i)+w_1(2-i)+2i-2 = (w_1(2-i)+(i-2))+i$. Note that the first equality of polynomials is a tautology, while the second is not.

It can be proved that an assertion is true on all paths iff whenever values from $\{0, 1\} \subseteq \mathbb{F}_p$ are substituted for the w_i's in the polynomials corresponding to the assertions, the resulting polynomials (in program's input variables) are equal over $\mathbb{F}_p$. The interesting aspect is that the polynomials corresponding to the valid assertions remain equal, even if the w_i's are treated as indeterminates (along with the program's input variables) over $\mathbb{F}_p$ since these polynomials are multilinear in w_i's, meaning that no w_i occurs with an exponent greater than 1.

The significance of reducing the problem to that of detecting polynomial equality over variables that take values from a large field lies in the following classic theorem due to Schwartz and Zippel [Sch80, Zip79].

Theorem 7 [Randomized Polynomial Identity Testing] *Let $Q_1(x_1, .., x_m)$ and $Q_2(x_1, .., x_m)$ be two different multivariate polynomials of degree at most d, in variables $x_1, .., x_m$ over some field $\mathbb{F}$. Fix any finite set $A \subseteq \mathbb{F}$, and let $r_1, .., r_m$ be chosen independently and uniformly at random from A. The probability that this choice is such that $Q_1(r_1, .., r_m) = Q_2(r_1, .., r_m)$ is at most $\frac{d}{|A|}$.*

Schwartz and Zippel's theorem says that a random evaluation of two different polynomials will quite likely yield different results. The theorem suggests that the error probability in the random interpretation scheme can be reduced by increasing the size of the set from which the random values are chosen. Additionally, the error probability decreases exponentially with the number of independent trials. Random testing can be thought of as an instance of this random interpretation scheme wherein the choice of weights w is restricted to the small set $\{0, 1\}$ (this corresponds to executing either the true branch or the false branch of a conditional); but this gives a useless bound of $d/2$ for the error probability.

Note that when fully expanded, the polynomials corresponding to the assertions could be exponential in size of the program; however, this is not a problem since we only need to evaluate them and our interpreter can evaluate them in time linear in size of the program.

The lack of a known polynomial time deterministic algorithm for checking the equality of polynomials (presented as straight-line programs) suggests that randomization has a chance to surpass deterministic algorithms in those program analysis problems that can be naturally reduced to checking equality of polynomials. Therefore it is not surprising that random interpretation works so well for checking equalities in programs that involve only linear arithmetic computations. We show in the next chapter that even some non-arithmetic operators can be encoded using polynomials. These schemes are however not as obvious as for linear arithmetic. We then formalize this notion in Section 4.1.1.

2.4 Related Work

There are a couple of algorithms that have been proposed in the literature for discovering linear equalities among program variables.

Karr's algorithm [Kar76] matches the precision of the algorithm that we have described in this chapter. It performs abstract interpretation over the lattice of linear equalities. However, its transfer functions are complicated and expensive. The transfer functions for the assignment node and conditional node take $O(k_v^2)$ time, while the transfer function for the join node, which requires computing the union of affine subspaces, takes $O(k_v^3)$ time. In contrast, our algorithm performs random interpretation over the same lattice of linear equalities. It uses simpler and efficient operations. It requires $O(k_v)$ time for processing an assignment node, and $O(k_v^2)$ time for processing a conditional and a join node. Karr did not prove any bound on the size of the numbers that may arise during the computation. Since the transfer functions in the Karr's algorithm involve multiplying two numbers of the same size, an implementation of Karr's algorithm may have to deal with numbers that require exponential (in the size of the program) number of bits for representation. Our algorithm also involves multiplying two numbers of the same size, but it is able to avoid the problem of dealing with exponentially large numbers by using the strategy of performing arithmetic modulo a randomly chosen prime number, which in the worst case requires bits that are linear in the size of the program and the number of program variables. Karr's algorithm can also employ the same strategy, but then it no longer remains a deterministic algorithm.

Cousot and Halbwachs' algorithm [CH78] discovers linear inequality relationships among variables. Since linear equalities are a special case of linear inequalities, their algorithm is more precise than our algorithm or Karr's algorithm. However, it is correspondingly even more complicated and expensive.

Recently, Müller-Olm and Seidl [MOS04a] specialized Karr's algorithm for the case of non-deterministic conditionals. Instead of using linear equalities to represent an affine space (as is the case in Karr's algorithm), they use linearly independent vectors to represent an affine space. Furthermore, using the strategy of semi-naive fixpoint iteration [FS98], they are able to improve the complexity of Karr's algorithm to $O(nk_{\mathtt{v}}^3)$ for the case of non-deterministic conditionals (assuming unit cost for arithmetic operations). Our algorithm when specialized to the case of non-deterministic conditionals has a complexity of $O(n_{\mathtt{s}}k_{\mathtt{u}}t)$ (assuming that the maximum number of back-edges β in any program loop is a small constant, as discussed in Section 2.3). The quantity t, which denotes the number of states in each sample computed by the random interpreter, is a small constant if the goal is to verify linear equalities or to discover linear equalities that involve a constant number of program variables. If the goal is to discover linear equalities that involve potentially all program variables, then $t = O(k_{\mathtt{v}})$.

Chaos umpire sits,
And by decision more embroils the fray
By which he reigns: next him high arbiter
Chance governs all.

John Milton, *Paradise Lost.*

Chapter 3

Uninterpreted Functions

In this chapter, we consider the problem of discovering equivalences between program sub-expressions. This problem is also known as *global value numbering* for historical reasons. The general problem of discovering equivalences among program sub-expressions is undecidable. Hence, we make the following two approximations.

We consider programs that have been abstracted using uninterpreted functions (i.e., each distinct program operator is represented by a distinct uninterpreted function). An uninterpreted function F^a of arity a satisfies only one axiom: If equal arguments are passed to two function applications, then equal result is obtained:

$$\left(\bigwedge_{i=1}^{a} e_i = e_i' \right) \Rightarrow F^a(e_1, \ldots, e_a) = F^a(e_1', \ldots, e_a') \tag{3.1}$$

The above axiom is also called the *congruence axiom*. In other words, F^a is an unknown and a side-effect free function. Uninterpreted functions are a commonly used abstraction to reason about program operators that are hard to reason about precisely. For example, consider the programs P_1 and P_2 shown in Figure 3.1. Program P_1 has two assertions at the end, and both of them are true. In general, reasoning about multiplication (in presence of addition and over integers) is undecidable. Hence, a commonly used technique is to abstract away multiplication by an uninterpreted function. Program P_2 has been obtained from program P_1 by abstracting the multiplication operator in program P_1 using a binary uninterpreted function F. The disadvantage of such an abstraction is loss of precision; note that program P_2 satisfies only the first assertion. It does not satisfy the second assertion

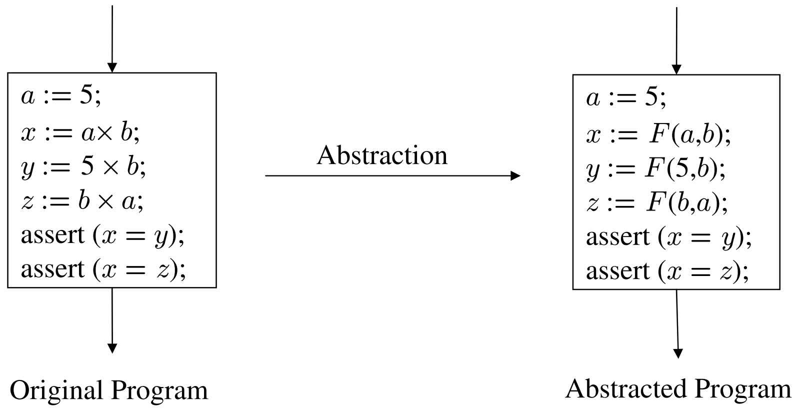


Figure 3.1: Example of program abstraction using uninterpreted functions.

since the uninterpreted function does not obey the commutativity axiom. However, the advantage is that reasoning about uninterpreted functions is easier than multiplication operator. This form of equivalence, where the operators are treated as uninterpreted functions, is called *Herbrand equivalence.*

We also assume that all conditionals in the program are non-deterministic. Without this assumption, the problem is undecidable [MORS05].

We present a polynomial time randomized algorithm that discovers all equivalences among program sub-expressions under the above two assumptions about the program abstraction [GN04a]. This program abstraction is described more formally in Section 3.2. Earlier algorithms for this problem were either exponential, or incomplete, i.e., they did not discover all equivalences among program sub-expressions (even under the same assumptions of program operators being modeled as uninterpreted functions and conditionals being treated as non-deterministic).

Applications: Detecting equivalence of program sub-expressions is a prerequisite for many important optimizations like constant and copy propagation [BA98, WZ91], common sub-expression elimination, invariant code motion [Cli95, RWZ88], induction variable elimination, branch elimination, branch fusion, and loop jamming [Muc00]. It is also important for discovering equivalent computations in different programs, for example, plagiarism detection and translation validation [PSS98, Nec00], where a program is compared with the optimized version in order to check the correctness of the optimizer.

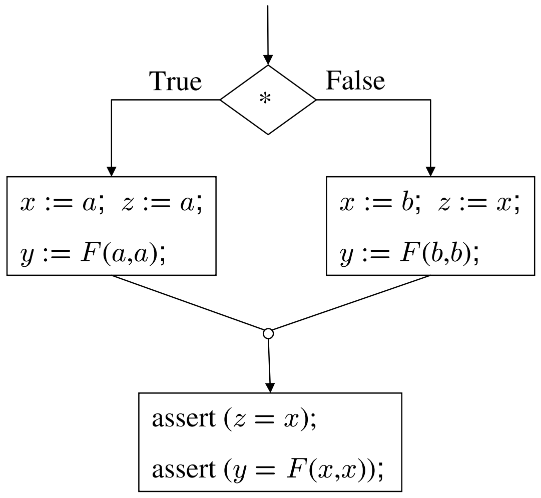


Figure 3.2: Example of non-trivial assertions in a program abstracted using uninterpreted functions.

3.1 Key Ideas

We first give some intuition as to why reasoning about equivalences of expressions involving uninterpreted functions in a program is non-trivial. Suppose the program expressions belong to the following language of expressions, where F is a binary uninterpreted function, and x denotes some program variable.

$$e ::= x \quad | \quad F(e_1, e_2)$$

Two expressions in the above language are equal iff they are syntactically equal. Hence, reasoning about equivalences of such expressions in a straight-line program is easy: compute the symbolic values of all expressions (in terms of uninterpreted functions and input variables) and check for syntactic equality. The difficulty comes in presence of conditionals and join points in programs when the symbolic values of expressions cannot be expressed using simply uninterpreted functions and input variables. For example, consider the program shown in Figure 3.2. The symbolic value of x is either a or b depending upon whether the conditional evaluates to true or false, and hence cannot be expressed using only uninterpreted functions and input variables. However, we can express the symbolic value of x using the standard ϕ functions used in SSA form. (A ϕ function should not be confused with the affine join operator ϕ_w, which takes a weight w as a subscript.)

$$x = \phi(a, b) \tag{3.2}$$

Similarly, the symbolic values of z and y are:

$$z = \phi(a, b) \tag{3.3}$$

$$y = \phi(F(a, a), F(b, b)) \tag{3.4}$$

The symbolic value for $F(x, x)$ can be obtained by application of F to the symbolic value of x.

$$F(x, x) = F(\phi(a, b), \phi(a, b)) \tag{3.5}$$

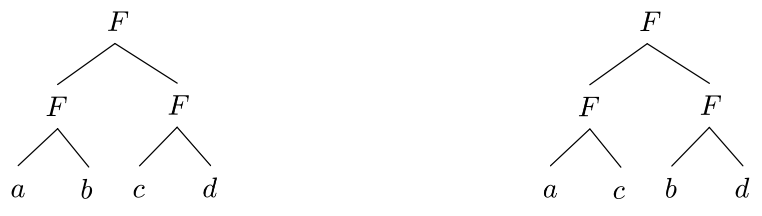
The symbolic values of x and z are syntactically identical. However, the symbolic values of y and $F(x, x)$ are not syntactically identical even though $y = F(x, x)$. This is because a ϕ function is not an uninterpreted function; it is more than an uninterpreted function. It satisfies the following two axioms:

$$\phi(e, e) = e \tag{3.6}$$

$$\phi(F(e_1, e_2), F(e'_1, e'_2)) = F(\phi(e_1, e'_1), \phi(e_2, e'_2)) \tag{3.7}$$

Alpern, Wegman, and Zadeck's (AWZ) algorithm [AWZ88] reasons about equivalences of program sub-expressions by computing the symbolic values of expressions as described above. (The AWZ algorithm is described in more detail in Section 3.4.) It treats ϕ functions as uninterpreted functions. Hence, it is able to prove that $z = x$. However, since it does not capture the semantics of ϕ functions, it is not able to prove that $y = F(x, x)$.

Our algorithm for reasoning about equivalences of program sub-expressions is based on the idea of random interpretation, which relies on executing a program on a number of random inputs and discovering relationships from the computed values. Both branches of a conditional are executed and at join points the program states are combined using a random affine combination. Note that our algorithm tries to capture the semantics of ϕ functions by giving them an affine join interpretation. An affine join interpretation of ϕ functions clearly satisfies Equation 3.6. The challenging part now is to identify how to execute expressions involving uninterpreted functions such that Equation 3.7 is also satisfied (under the affine join interpretation of ϕ functions).



Expression e Expression e'

$$\begin{aligned} I(e) &= F(F(a,b),F(c,d)) \\ &= r_1[r_1(a) + r_2(b)] + r_2[r_1(c) + r_2(d)] \\ &= r_1^2(a) + r_1 r_2(b+c) + r_2^2(d) \end{aligned}$$

$$\begin{aligned} I(e) &= F(F(a,c),F(b,d)) \\ &= r_1[r_1(a) + r_2(c)] + r_2[r_1(b) + r_2(d)] \\ &= r_1^2(a) + r_1 r_2(c+b) + r_2^2(d) \end{aligned}$$

Figure 3.3: Example of two distinct uninterpreted function terms e and e' that are equal when the binary uninterpreted function F is modeled as a linear function of its arguments.

We use the idea of choosing random interpretations for the uninterpreted functions. For example, let us choose the following class of non-linear random interpretations I for uninterpreted functions, where r_1 and r_2 are some random integers.

$$\begin{aligned} I(F(e_1, e_2)) &= r_1 I(e_1)^2 + r_2 I(e_2)^2 \\ I(x) &= x \end{aligned}$$

The above interpretation I is good for preserving equivalences of expressions in straight-line procedures. However, it does not satisfy Equation 3.7 in presence of the affine join interpretation for ϕ functions. Similar problems arise for any non-linear interpretation I. This is because the affine join operation does not preserve non-linear equalities (as pointed out on page 12 in Section 2.1.1).

Let us now consider random linear interpretations. For example, let us choose the following class of random interpretations I for uninterpreted functions, where r_1 and r_2 are some random integers.

$$\begin{aligned} I(F(e_1, e_2)) &= r_1 I(e_1) + r_2 I(e_2) \\ I(x) &= x \end{aligned}$$

The above interpretation I does satisfy Equation 3.6 in presence of the affine join interpretation for ϕ functions. However, it introduces false equivalences even in straight-line programs. For example, consider the two distinct expressions e and e' in Figure 3.3, represented as trees. Note that $I(e) = I(e')$ under the above class of interpretations I. Similar problems arise for any linear interpretation I.

It appears that we have reached an impasse (if we fix the affine join interpretation for ϕ functions). If we choose non-linear interpretations for uninterpreted function F, then we do not preserve desired equivalences across a join point. If we choose linear interpretations for uninterpreted function F, then we introduce undesirable equivalences.

3.1.1 Random Linear Interpretation over Vectors

One way to characterize the failure of soundness (i.e., distinct expressions mapping to equal expressions) when using linear interpretations is the commutativity of scalar multiplication. Note that in Figure 3.3, the monomials r_1r_2b and r_2r_1b are equal, and so are the monomials r_2r_1c and r_1r_2c. This leads to $I(e) = I(e')$ even though $e \neq e'$. We can potentially avoid this problem by performing multiplication in a non-commutative structure such as matrices. Similar ideas have been used in algorithms for non-commutative polynomial identity testing [BW05].

Another way to characterize the failure of soundness when using linear interpretations is that we are restricted to at most three [1] random coefficients, which are too few to encode a large number of leaves. Thus, it is possible for two distinct trees to have identical interpretations. To increase the number of coefficients while maintaining linearity, we may consider maintaining more than one (scalar) value for each variable and thus each expression. This will enable us to introduce more random parameters in the interpretation function.

Inspired by the above observations, we propose giving random linear interpretations to uninterpreted functions over vectors and matrices. Each expression e is mapped to a value $I(e)$ that is a vector of ℓ integers, using the following interpretation for uninterpreted functions.

$$I(F(e_1, e_2)) = R_1 I(e_1) + R_2 I(e_2)$$
$$I(x) = x$$

[1]The most general linear mapping I for a binary uninterpreted function $F(e_1, e_2)$ is $I(F(e_1, e_2)) = r_1 I(e_1) + r_2 I(e_2) + r_3$, which uses three random coefficients r_1, r_2, and r_3. Even with this mapping, $I(e) = I(e')$, where e and e' are the distinct expressions shown in Figure 3.3.

Here R_1 and R_2 are some randomly chosen $\ell \times \ell$ matrices. ℓ is a parameter of the random interpreter, and we will derive lower bounds for ℓ later in this section.

The random matrices R_1 and R_2 can be sparse as shown below; this allows for efficient computation of the vector value for $F(e_1, e_2)$ given the vector values for e_1 and e_2.

$$R_1 = \begin{pmatrix} r_1 & 0 & 0 & \dots & 0 & s_\ell \\ s_1 & r_2 & 0 & \dots & 0 & 0 \\ 0 & s_2 & r_3 & \dots & 0 & 0 \\ & & & \ddots & & \\ 0 & 0 & 0 & \dots & r_{\ell-1} & 0 \\ 0 & 0 & 0 & \dots & s_{\ell-1} & r_\ell \end{pmatrix} \text{ and } R_2 = \begin{pmatrix} r'_1 & 0 & 0 & \dots & 0 & s'_\ell \\ s'_1 & r'_2 & 0 & \dots & 0 & 0 \\ 0 & s'_2 & r'_3 & \dots & 0 & 0 \\ & & & \ddots & & \\ 0 & 0 & 0 & \dots & r'_{\ell-1} & 0 \\ 0 & 0 & 0 & \dots & s'_{\ell-1} & r'_\ell \end{pmatrix}$$

Above, r_i, r'_i, s_i, s'_i are all randomly chosen integers.

Lemma 7 (stated below) characterizes the desired property of the above choice of linear interpretations I for uninterpreted functions. Note that I uses the sparse random matrices R_1 and R_2 shown above (each with 2ℓ entries). Let `SEval`(e) denote the vector obtained by applying I to e, but treating the matrix entries r_i, r'_i, s_i, s'_i as variables rather than random integers. Note that each element in the vector `SEval`(e) is a polynomial in program variables as well as the variables r_i, r'_i, s_i, s'_i.

Lemma 7 [Linear Interpretation Soundness Lemma] *Let e and e' be two tree expressions such that e has at most 2^ℓ leaves. If* `SEval`$(e) =$ `SEval`(e')*, then $e = e'$.*

Proof. For an expression e and index i between 1 and ℓ, let $Q(e, i)$ denote the i^{th} element in the vector `SEval`(e) after substituting $s_\ell = s'_\ell = 0$. Thus,

$$\begin{aligned} Q(x, i) &= x \\ Q(F(e_1, e_2), 1) &= r_1 Q(e_1, 1) + r'_1 Q(e_2, 1) \\ Q(F(e_1, e_2), i) &= r_i Q(e_1, i) + s_{i-1} Q(e_1, i-1) + r'_i Q(e_2, i) + s'_{i-1} Q(e_2, i-1) \quad \text{for } \ i > 1 \end{aligned}$$

Note that for any i, $Q(e, i)$ does not contain any of the variables $r_{i+1}, \ldots, r_\ell$, $r'_{i+1}, \ldots, r'_\ell$, $s_i, \ldots, s_{\ell-1}$, and $s'_i, \ldots, s'_{\ell-1}$. This means that the polynomial $Q(F(e_1, e_2), i)$ can be decomposed uniquely into the subpolynomials $r_i Q(e_1, i) + r'_i Q(e_2, i)$ (all of whose monomials contain variable r_i or r'_i), $s_{i-1} Q(e_1, i-1)$ (all of whose monomials contain variable s_{i-1} but not r_i or r'_i), and $s'_{i-1} Q(e_2, i-1)$ (all of whose monomials contain variable s'_{i-1}

but not r_i, r'_i or s_{i-1}). This implies that for any $i > 1$, we have:

$$Q(F(e_1, e_2), i) = Q(F(e'_1, e'_2), i) \Rightarrow r_iQ(e_1, i) + r'_iQ(e_2, i) = r_iQ(e'_1, i) + r'_iQ(e'_2, i) \quad (3.8)$$

$$\text{and } Q(e_1, i-1) = Q(e'_1, i-1) \quad (3.9)$$

$$\text{and } Q(e_2, i-1) = Q(e'_2, i-1) \quad (3.10)$$

We now prove that if $Q(e, i) = Q(e', i)$, $i \geq j$ and e has at most 2^j leaves for some i and j, then $e = e'$. Note that the degree of polynomial $Q(e, i)$ is equal to the depth of expression e. Hence, if $Q(e, i) = Q(e', i)$, then $Q(e, i)$ and $Q(e', i)$ have the same degree, and hence e and e' have the same depth. The proof is by induction on the depth of e. The base case is trivial since e and e' are both leaves and $Q(e, i) = e$ and $Q(e', i) = e'$. For the inductive case, $e = F(e_1, e_2)$ and $e' = F(e'_1, e'_2)$. Suppose $Q(F(e_1, e_2), i) = Q(F(e'_1, e'_2), i)$ for some $i \geq j$. Since e has at most 2^j leaves, it must be that at least one of e_1 or e_2 has at most 2^{j-1} leaves. Consider the case when e_1 has at most 2^{j-1} leaves (the other case is symmetric). From Equation 3.9, we have that $Q(e_1, i-1) = Q(e'_1, i-1)$. Since $i-1 \geq j-1$, we can apply the induction hypothesis for e_1 and e'_1 to obtain that $e_1 = e'_1$. Consequently, $Q(e_1, i) = Q(e'_1, i)$. This allows us to simplify Equation 3.8 to $Q(e_2, i) = Q(e'_2, i)$. Since e_2 has at most 2^j leaves, we can apply the induction hypothesis for e_2 and e'_2 to conclude that $e_2 = e'_2$. This completes the proof. □

The above soundness lemma implies that `SEval` is an injective mapping from expressions to vectors of polynomials, and hence allows us to compare expressions e by random testing of their corresponding polynomials in `SEval`(e). Note that the larger the size ℓ of vectors `SEval`(e) is, the larger is the size of the expressions e that can be compared. The parameter ℓ must be at least the logarithm of the number of leaves in the tree representation of the expressions. Interestingly, this value does not depend on the depth of the expressions. A consequence is that expressions involving only unary constructors can be discriminated with $\ell = 1$, independent of their depth. The expressions involving binary uninterpreted functions and computed by a straight-line program are representable using DAGs (directed acyclic graphs) of size linear in the size of the program. The number of leaves in the tree representation of such expressions may be exponential in the depth of those expressions in the worst case. Thus, in order to distinguish such expressions, ℓ must be chosen to be the depth of those expressions in the worst case. We have performed a number of experiments that suggest that an even tighter bound on ℓ might be possible, but we are not able to prove

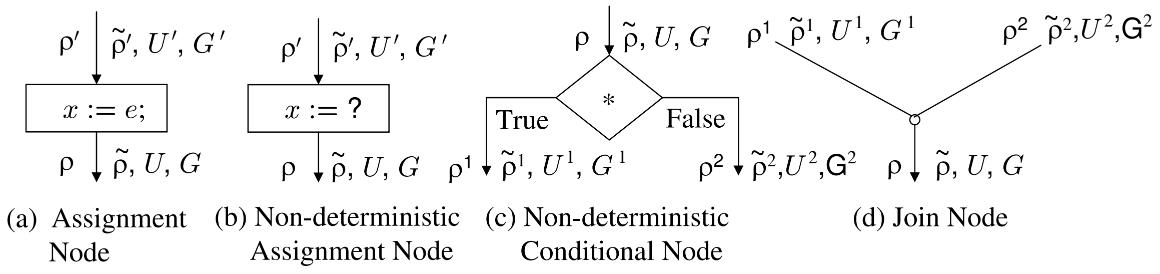


Figure 3.4: Flowchart nodes considered in the uninterpreted functions analysis.

any such result at the moment. (We feel that it may be possible to prove a result similar to Lemma 7 for the DAG representation of expressions too. Using less sparse matrices R_1 and R_2 may help towards proving any such result.)

3.2 The Random Interpreter

We assume that each procedure is abstracted using the flowchart nodes shown in Figure 3.4. Related to this program model, we continue to use the notation established in Section 2.2. The expression e here denotes an uninterpreted function term constructed from the following expression language:

$$e ::= x \quad | \quad F(e_1, e_2)$$

Here F is a binary uninterpreted function. Note that this is an intra-procedural analysis. The procedure calls have to be abstracted using non-deterministic assignments.

For simplicity, we consider only one binary uninterpreted function F above. The analysis can be easily extended to incorporate any number of uninterpreted functions of any arity. Alternatively, we can model any uninterpreted function F^a of any constant arity a using the given binary uninterpreted function F by employing the following closure trick:

$$F^a(e_1, \ldots, e_a) = F(e_1, e'_2), \text{ where } e'_i = \begin{cases} F(e_i, e'_{i+1}) & \text{for } 2 \le i \le a-1 \\ F(e_a, x_{F^a}) & \text{for } i = a \end{cases}$$

Here x_{F^a} is a fresh variable (can be regarded as a new input variable) associated with the uninterpreted function F^a. If we regard a to be a constant, then this modeling does not alter the quantities (except by a constant factor) such as n_s or k_u on which the computational complexity of the algorithm depends.

3.2.1 Basic Algorithm

The random interpreter executes each procedure in the program like an abstract interpreter or a data-flow analyzer. It maintains for each program point a state that maps program variables to (column) vectors with ℓ entries from the field $\mathbb{F}_p$. These states encode Herbrand equivalences, or equivalences among uninterpreted function terms of the program. Theorem 10 (on page 60) specifies the error probability of the random interpreter as a function of the parameters ℓ and p. The number of entries in each vector can be chosen to be $\ell = d_m(k_{\mathtt{u}} + 1)\beta + d_e$, where d_m is the maximum depth of any expression computed by the program along any acyclic path without taking any back-edge [2] and d_e is the maximum depth of expressions whose equivalences we want to decide. However, experiments suggest that even $\ell = \log(d_m(k_{\mathtt{u}} + 1)\beta + d_e)$ does not yield any error in practice. The prime p can be chosen to be any 32-bit prime.

A state at a program point is obtained from the state(s) at the immediately preceding program point(s). In presence of loops, the random interpreter goes around each loop until a fixed point is reached. The criterion for a fixed point is defined in Section 3.2.3. We now describe the action of the random interpreter on the flowchart nodes shown in Figure 3.4.

Initialization: At procedure entry, the random interpreter starts with an initial state ρ_0 that assigns to each variable x the column vector $(r_x, \ldots, r_x)^T$ (i.e., transpose of the vector $(r_x, \ldots, r_x)$), where r_x is an element chosen u.a.r. from $\mathbb{F}_p$.

$$\rho_0(x) = (r_x, \ldots, r_x)^T, \text{ where } r_x = \mathtt{Rand}()$$

Note that all elements of the vector $\rho_0(x)$ are assigned the same random value r_x. However, for each variable x, the random value r_x is chosen independently of the random choice r_y for any other variable y. The random interpreter initializes the states at all other program points to $\perp$.

Assignment Node: See Figure 3.4 (a).
If the state ρ' before the assignment node is $\perp$, then the state ρ after the assignment node is also $\perp$. Else, the random interpreter transforms the state ρ' before an assignment node

[2]Clearly, d_m is bounded above by the maximum number of occurrences of the uninterpreted function symbol F in any procedure.

$x := e$ by setting x to the value of e in that state, which is denoted by $\texttt{Eval}(e, \rho')$.

$$\rho = \rho'[x \leftarrow \texttt{Eval}(e, \rho')]$$

The random interpreter evaluates an expression e in a state ρ by giving random linear interpretations to the uninterpreted function F as described in Section 3.1.1.

$$\begin{aligned} \texttt{Eval}(x, \rho) &= \rho(x) \\ \texttt{Eval}(F(e_1, e_2), \rho) &= R_1 \times \texttt{Eval}(e_1, \rho) + R_2 \times \texttt{Eval}(e_2, \rho) \end{aligned}$$

R_1 and R_2 are the random sparse matrices shown in Section 3.1.1. These are chosen once at the start, and the same matrices are used for all occurrences of the uninterpreted function F. The $\times$ operator above refers to the multiplication of a $\ell \times \ell$ matrix by a column vector of size ℓ. The $+$ operator above refers to the addition of two column vectors of size ℓ. All operations are performed over the field $\mathbb{F}_p$.

If the expression language e consists of other uninterpreted functions, the $\texttt{Eval}$ function can be defined as follows. For a unary function F^1, $\texttt{Eval}(F^1(e), \rho)$ can be defined to be $R_1 \times \texttt{Eval}(e, \rho) + R_2$, while for any a-ary uninterpreted function F^a for $a > 1$, $\texttt{Eval}(F^a(e_1, \ldots, e_a), \rho)$ can be defined to be $\sum_{i=1}^{a} R_i \times \texttt{Eval}(e_i, \rho)$. For each distinct uninterpreted function symbol, a different set of random sparse matrices R_i are chosen. However, the same set of random matrices is used for all occurrences of a particular uninterpreted function.

Non-deterministic Assignment Node: See Figure 3.4 (b).
If the state ρ' before the assignment node is $\perp$, then the state ρ after the assignment node is also $\perp$. Else, the random interpreter processes the assignment $x :=?$ by transforming the state ρ' by setting x to some fresh random value.

$$\rho_i = \rho'_i[x \leftarrow [r, \ldots, r]^T], \text{ where } r = \texttt{Rand}()$$

Non-deterministic Conditional Node: See Figure 3.4 (c).
The random interpreter simply copies the state before the conditional on the two branches of the conditional.

$$\rho^1 = \rho \quad \text{and} \quad \rho^2 = \rho$$

Join Node: See Figure 3.4 (d).

If any one of the states before a join node is $\perp$, the random interpreter assigns the other state before the join node to the state after the join node. Else, the random interpreter chooses w u.a.r from $\mathbb{F}_p$ and performs an affine join of the two states ρ^1 and ρ^2 before the join node to obtain the state ρ after the join node.

$$\rho = \phi_w(\rho^1, \rho^2), \text{ where } w = \texttt{Rand}()$$

Note that by definition of ϕ_w, we have $\rho(x) = w \times \rho^1(x) + (1 - w) \times \rho^2(x)$. Here $\times$ operator refers to the multiplication of the column vector $\rho^1(x)$ by the scalar w, and the $+$ operator refers to the addition of two column vectors (over the field $\mathbb{F}_p$).

Verifying or Discovering Equal Program Sub-expressions

After fixed-point computation, the results of the random interpreter can be used to verify or discover equivalences among program sub-expressions at any program point as follows. Let ρ be the state computed by the random interpreter at program point π. If $\rho = \perp$, the random interpreter declares program point π to be unreachable. Else, it declares expressions e_1 and e_2 to be equal at program point π iff $\texttt{Eval}(e_1, \rho) = \texttt{Eval}(e_2, \rho)$.

Optimization

Maintaining a state explicitly at each program point is expensive (in terms of time and space complexity) and redundant. The optimization of maintaining one global state for the SSA version of the program, as discussed in Section 2.2.1, applies here as well.

3.2.2 Error Probability Analysis

For the purpose of the random interpreter's analysis, we introduce two new interpreters: a symbolic random interpreter, which is a symbolic version of the random interpreter, and an abstract interpreter that is known to be complete and sound. We prove that the symbolic random interpreter is as complete and as sound as the abstract interpreter. We then show that this implies that the random interpreter is complete and probabilistically sound.

The Symbolic Random Interpreter

The symbolic random interpreter maintains symbolic states $\tilde{\rho}$ and executes a program like the random interpreter but symbolically. Instead of using random values for the initial values for variables, or the parameters w, r_i, r'_i, s_i, and s'_i, it uses variable names and maintains symbolic expressions. We write $A(e, \tilde{\rho})$ to denote the symbolic value of expression e in symbolic state $\tilde{\rho}$. We also write $\tilde{\rho} \models e_1 = e_2$ when $A(e_1, \tilde{\rho}) = A(e_2, \tilde{\rho})$. We use the notation $\mathit{Degree}(A(e, \tilde{\rho}))$ to refer to the degree of the symbolic polynomials that are the elements of the symbolic vector $A(e, \tilde{\rho})$ while ignoring the contribution of the weight variables w to the degree. Note that all these polynomials have the same degree.

The following property states the relationship between the states computed by the random interpreter and the symbolic states computed by the symbolic random interpreter.

Property 2 *Let $\tilde{\rho}$ be a symbolic state computed by the symbolic random interpreter at some program point π. Let ρ be the corresponding state computed by the random interpreter at π. The state ρ can be obtained from the symbolic state $\tilde{\rho}$ by substituting the input variables, the weight and parameter variables w, r_i and s_i with the values that the random interpreter has chosen for them.*

The Abstract Interpreter

The abstract interpreter computes a set of Herbrand equivalences U at each program point. We write $U \Rightarrow e_1 = e_2$ to say that the conjunction of Herbrand equivalences in U implies $e_1 = e_2$. We write $U^1 \cap U^2$ for the set of Herbrand equivalences that are implied by both U^1 and U^2. Finally, we write $U[e/x]$ for the relationships that are obtained from those in U by substituting expression e for variable x. For notational convenience, we let $\perp$ represent an inconsistent set of Herbrand equivalences. We say that $\perp \Rightarrow e_1 = e_2$ for all expressions e_1 and e_2. With these definitions we now define the action of the abstract interpreter over the flowchart nodes shown in Figure 3.4.

Initialization: The abstract interpreter initializes U to the empty set of Herbrand equivalences at procedure entry. At all other points, it initializes U to $\perp$.

Assignment Node: See Figure 3.4 (a).
If $U' = \perp$, then $U = \perp$. Else, $U = \{x = e[x'/x]\} \cup U'[x'/x]$, where x' is a fresh variable.

Non-deterministic Assignment Node: See Figure 3.4 (b).
If $U' = \bot$, then $U = \bot$. Else, $U = U'[x'/x]$, where x' is a fresh variable.

Non-deterministic Conditional Node: See Figure 3.4 (c).
$U^1 = U$ and $U^2 = U$.

Join Node: See Figure 3.4 (d).
If $U^1 = \bot$, then $U = U^2$. Else, if $U^2 = \bot$, then $U = U^1$. Else, $U = U^1 \cap U^2$.

Implementations of the abstract interpreter defined above have been described in the literature [Kil73]. The major concern there is the concrete representation of the set U and the implementation of the operation $U^1 \cap U^2$. In Kildall's original presentation, the set U has an exponential-size representation, although this is not necessary [RKS99]. Here we use the abstract interpreter only to state and prove the completeness and soundness results of the random interpreter. The abstract interpreter is sound and complete when all operators are uninterpreted and conditionals are non-deterministic [Ste87, RKS99].

We now state the relationship between the symbolic states $\tilde{\rho}$ computed by the symbolic random interpreter and the Herbrand equivalences U computed by the abstract interpreter in the form of completeness and soundness theorems.

Theorem 8 [Completeness Theorem] *Let U be a set of Herbrand equivalences computed by the abstract interpreter at some program point π. Let $\tilde{\rho}$ be the corresponding symbolic state computed by the symbolic random interpreter at π. Let e_1 and e_2 be any two expressions such that $U \Rightarrow e_1 = e_2$. Then, $\tilde{\rho} \models e_1 = e_2$.*

The completeness theorem implies that the symbolic random interpreter (and hence the random interpreter) discovers all the Herbrand equivalences that the abstract interpreter discovers. The proof of Theorem 8 is based on Lemma 8 which is stated and proved below. Lemma 8 states that the affine join of two states $\tilde{\rho}_1$ and $\tilde{\rho}_2$ satisfies all the Herbrand equivalences that are satisfied by both the states $\tilde{\rho}_1$ and $\tilde{\rho}_2$. The full proof of Theorem 8 is given in Appendix B.1.

Lemma 8 [Affine Join Completeness Lemma] *Let $\tilde{\rho}_1$ and $\tilde{\rho}_2$ be two symbolic states that satisfy the Herbrand equivalence $e_1 = e_2$. Then, for any choice of weight w, $\phi_w(\tilde{\rho}_1, \tilde{\rho}_2)$, which is the affine join of $\tilde{\rho}_1$ and $\tilde{\rho}_2$, also satisfies the same Herbrand equivalence $e_1 = e_2$.*

Proof. Note that for any expression e, and any symbolic state $\tilde{\rho}$, all elements of the vector $A(e, \tilde{\rho})$ are a linear function of the program variables in expression e. Hence, for any affine combination $\phi_w(\tilde{\rho}_1, \tilde{\rho}_2)$ of two symbolic states $\tilde{\rho}_1$ and $\tilde{\rho}_2$, it can be easily verified that $A(e, \phi_w(\tilde{\rho}_1, \tilde{\rho}_2)) = w \times A(e, \tilde{\rho}_1) + (1 - w) \times A(e, \tilde{\rho}_2)$. It thus follows that if $A(e_1, \tilde{\rho}_1) = A(e_2, \tilde{\rho}_1)$ and $A(e_1, \tilde{\rho}_2) = A(e_2, \tilde{\rho}_2)$, then $A(e_1, \phi_w(\tilde{\rho}_1, \tilde{\rho}_2)) = A(e_2, \phi_w(\tilde{\rho}_1, \tilde{\rho}_2))$. From here the completeness lemma follows immediately. $\square$

It is not surprising that the completeness lemma holds, since we have chosen the linear interpretations of operators specifically to satisfy this constraint. Next we state the soundness theorem.

Theorem 9 [Soundness Theorem] *Let U be a set of Herbrand equivalences computed by the abstract interpreter at some program point π. Let $\tilde{\rho}$ be the corresponding symbolic state computed by the symbolic random interpreter at π. Let e_1 and e_2 be two expressions such that $\tilde{\rho} \models e_1 = e_2$, and $\ell \geq Degree(A(e_1, \tilde{\rho}))$. Then, $U \Rightarrow e_1 = e_2$.*

According to Theorem 9, if the symbolic polynomials associated with two expressions under our random interpretation scheme are equal, then those two expressions are also found equal by the abstract interpreter. The proof of Theorem 9 is based on Lemma 7 (on page 7). Notice, however, that in Theorem 9 the lower bound on ℓ is stated based on the degree of $A(e, \tilde{\rho})$, which is equal to the depth of expression e, while in Lemma 7, it is based on the logarithm on the number of leaves. The reason for this weakening of the soundness statement is two-fold: it would have been more complicated to carry out the proof with leaf counts, and in the worst case these measures are equal. The full proof of Theorem 9 (which is by induction on the number of operations performed by the interpreters) is given in Appendix B.2.

It follows from the discussion after Theorem 11 in the next section that the random interpreter goes around each loop at most $(k_{\rm u} + 1)\beta$ times for fixed-point computation¡. We use this observation along with Theorem 9 to prove Theorem 10, which establishes an upper bound on the probability that the random interpreter is unsound.

Theorem 10 [Probabilistic Soundness Theorem] *Let e_1 and e_2 be two unequal expressions of depth at most d_e at some program point π. Let d_m be the maximum depth of any expression computed by the program along any acyclic path without taking any back-*

edge. Let ρ be the random state computed by the random interpreter at π after fixed-point computation. If $\ell \geq d_m(k_{\mathtt{u}}+1)\beta + d_e$, then $\Pr[\rho \models e_1 = e_2] \leq \frac{(d_m+n_{\mathtt{c}})(k_{\mathtt{u}}+1)\beta+d_e}{p}$.

Proof. Let $\tilde{\rho}$ be the corresponding symbolic state, and U be the corresponding set of Herbrand equivalences at π. Since the abstract interpreter is sound, $U \not\Rightarrow e_1 = e_2$. It follows from the discussion after Theorem 11 that the random interpreter goes around each loop at most $(k_{\mathtt{u}}+1)\beta$ times for fixed-point computation. Hence, $\mathit{Degree}(A(e_1, \tilde{\rho})) \leq d_m(k_{\mathtt{u}}+1)\beta + d_e$. It thus follows from Theorem 9 that $\tilde{\rho} \not\models e_1 = e_2$. Note that the degree of the polynomials that are elements of the vector $A(e_1, \rho)$ are at most $(d_m+n_{\mathtt{c}})(k_{\mathtt{u}}+1)\beta+d_e$. The desired result now follows from Property 2 and Theorem 7. □

Theorem 10 implies that by choosing p big enough, the error probability can be made as small as we like. In particular, if $(d_m + n_{\mathtt{c}})(k_{\mathtt{u}} + 1)\beta + d_e < 10^3$, and if we choose $q \approx 2^{32}$ (which means that the random interpreter can perform arithmetic using 32-bit numbers), then the error probability is bounded above by 10^{-6}. By repeating the algorithm m times, the error probability can be further reduced to 10^{-6m}.

3.2.3 Fixed-point Computation

In presence of loops in procedures, the random interpreter goes around loops just like any abstract interpreter or a data-flow analyzer until a fixed point is reached. We say that the random interpreter reaches a fixed point across a loop when the sets of Herbrand equivalences (computed by the abstract interpreter defined earlier) corresponding to the states computed by the random interpreter reach a fixed point. Note that the states computed by the random interpreter themselves do not reach a fixed point, but the Herbrand equivalences represented by them do.

The elements of the abstract lattice over which the abstract interpreter performs computations are sets of Herbrand equivalences between program variables. These elements are ordered by the implication relationship (i.e., if U^2 is above U^1 in the abstract lattice, then $U^1 \Rightarrow U^2$). The following theorem provides a bound on the number of iterations required to reach a fixed point across a loop.

Theorem 11 [Fixed Point Theorem] *Let $U^1, \ldots, U^m$ be the sets of Herbrand equivalences that are computed by the abstract interpreter at some point π inside a loop in succes-*

sive iterations of that loop such that $U^i \not\equiv U^{i+1}$. Then, $m \leq k_{\mathbf{u}} + 1$, where $k_{\mathbf{u}}$ is the number of variables that are visible at π as well as modified inside that loop.

The proof of Theorem 11 follows from Lemma 9 and Property 3 stated below.

We first introduce some notation before stating the desired results. For any expression e, we use the notation $\mathit{Vars}(e)$ to denote the set of variables that occur in expression e. Let $\preccurlyeq$ denote any total ordering on all program variables. We use the notation $x \prec y$ to denote that $x \preccurlyeq y$, and that x and y are distinct variables. For notational convenience, we say that for any variable x, and expressions e_1 and e_2, $F(e_1, e_2) \prec x$.

Lemma 9 *Let V_π denote the set of variables that are visible at program point π. The Herbrand equivalences at program point π can be represented by a pair $H = (V, E)$, where $V \subseteq V_\pi$ is a set of independent variables and E is a set of equivalences $x = e$, one for each variable $x \in V_\pi - V$, such that $\mathit{Vars}(e) \subseteq V$. Furthermore, if $x \in V$, then $x \prec y$ for all variables y such that $(y = x) \in E$.*

The proof of this lemma is by induction on structure of the program and is given in Appendix B.3.

Property 3 *Let (V_1, E_1) and (V_2, E_2) represent the Herbrand equivalences computed by the abstract interpreter at any point inside a loop in two successive iterations of that loop. Suppose E_2 is a weaker set of equivalences than E_1. Then $V_2 \supset V_1$.*

Proof. We first make two useful observations. Let (V, E) represent the Herbrand equivalences at any program point. Then, (a) $E \not\Rightarrow x = e$ if $x \in V$, and $e \prec x$. (b) $E \not\Rightarrow e_1 = e_2$ if $\mathit{Vars}(e_1) \subseteq V$, $\mathit{Vars}(e_2) \subseteq V$ and $e_1 \not\equiv e_2$.

We first show that $V_2 \supseteq V_1$. Suppose for the purpose of contradiction that $V_2 \not\supseteq V_1$. Then, $E_2 \Rightarrow x = e$ for some variable $x \in V_1$ and expression e such that $e \prec x$. Since E_1 represents a stronger set of equivalences, $E_1 \Rightarrow x = e$. But this is not possible because of observation (a) above.

We now show that $V_2 \supset V_1$. Suppose for the purpose of contradiction that $V_2 = V_1$. Since E_1 is stronger than E_2, $E_1 \Rightarrow x = e_1$ for some $x \in V_\pi - V_1$ and expression e_1 such that $\mathit{Vars}(e_1) \subseteq V_1$ and $E_2 \not\Rightarrow x = e_1$. Note that $x \in V_\pi - V_2$ since $V_2 = V_1$. Hence, there exists an expression e_2 such that $E_2 \Rightarrow x = e_2$, where $\mathit{Vars}(e_2) \subseteq V_2$. Note that $e_1 \not\equiv e_2$

since $E_2 \not\Rightarrow x = e_1$ and $E_2 \Rightarrow x = e_2$, Since E_1 is stronger than E_2, $E_1 \Rightarrow x = e_2$ and hence $E_1 \Rightarrow e_1 = e_2$. But this is not possible because of observation (b) above. □

Note that if (V_1, E_1) and (V_2, E_2) represent the Herbrand equivalences computed by the abstract interpreter at some program point π inside a loop in two successive iterations, then $(V_2 - V_1)$ is a subset of the variables that are visible at π as well as modified inside the loop. The proof of Theorem 11 now follows from Lemma 9 and Property 3.

One way to detect when the random interpreter has reached a fixed point is to compare the sets of Herbrand equivalences implied by the random interpreter in two successive executions of a loop (this can be done by building a symbolic value flow graph of the program [RKS90], and then a disjoint partition of its nodes into congruence classes). If these sets are identical, then a fixed point for that loop has been reached. Computing the sets of Herbrand equivalences over the symbolic value flow graph takes time $O(n_{\mathtt{s}} \times \ell)$, and hence it may be an expensive operation. A better strategy may be to iterate around a loop (maximal strongly connected component) $(k_{\mathtt{u}} + 1)\beta$ times, where β is the number of back-edges in the loop. Note that it it guaranteed that a fixed point will be reached after iterating across a loop $(k_{\mathtt{u}} + 1)\beta$ times. This is because if a fixed point is not reached, then the set of Herbrand equivalences corresponding to at least one of the states (computed by the random interpreter) at the target of the back-edges must change, and it follows from Theorem 11 that there can be at most $k_{\mathtt{u}} + 1$ such changes at each program point.

3.2.4 Computational Complexity

We assume that the random interpreter performs the optimization of maintaining one global state in the SSA version of the program as discussed in Section 3.2.1. Under that optimization, a join operation reduces to processing phi-assignments at the corresponding join point. The cost of processing each assignment, both phi and non-phi, is $O(\ell)$. Computing the set of Herbrand equivalences from the global state for detecting a fixed point may be an expensive operation; hence it may be more efficient for the random interpreter to go around each loop $k_{\mathtt{u}}\beta$ times (as discussed after Theorem 5). Hence, the total number of operations performed by the random interpreter is bounded above by $(n_{\mathtt{s}} k_{\mathtt{u}} \beta)$. Assuming β to be constant, the running time of the random interpreter is $O(\ell n_{\mathtt{s}} k_{\mathtt{u}})$. Note that we choose ℓ to be greater than $d_m(k_{\mathtt{u}} + 1)\beta$ in order to satisfy the requirement for probabilistic soundness, where d_m is the maximum depth of any expression computed by the program along any acyclic path without taking any back-edge. Hence, this yields an overall complexity of $O(n_{\mathtt{s}}k_{\mathtt{u}}^2 d_m)$ for the random interpreter.

Our analysis for probabilistic soundness requires choosing ℓ to be greater than $d_m(k_{\mathtt{u}}+1)\beta$. However, we feel that our analysis is conservative. The experiments that we have performed also suggest that tighter bounds on ℓ might be possible, but we are not able to prove any such result at the moment. Note that Lemma 7 requires working with vectors of size only $\log t$, where t is the size of the tree expressions. If we can prove a similar lemma for DAGs, then we can prove that choosing $\ell = O(\log n_{\mathtt{s}})$ is sufficient for probabilistic soundness, which will yield an overall complexity of $O(n_{\mathtt{s}}k_{\mathtt{u}} \log n_{\mathtt{s}})$ for the random interpreter.

3.3 Deterministic Algorithm

In this section, we describe a polynomial-time deterministic algorithm, which is as precise as the randomized algorithm that we have described earlier in this chapter. This deterministic algorithm is however slightly less efficient than the randomized algorithm. This deterministic algorithm discovers all equivalences among program sub-expressions when all program operators have been abstracted away as uninterpreted functions, and all conditionals have been abstracted away as non-deterministic conditionals.

Earlier, there had been several attempts at developing a polynomial-time deterministic algorithm for this problem. For us, the inspiration to develop a polynomial-time deterministic algorithm actually came after developing the randomized algorithm. It is believed that randomization cannot yield an exponential-time speedup compared to deterministic algorithms; though it can yield algorithms that are more efficient than the best possible deterministic algorithm by a polynomial factor. After developing a polynomial-time randomized algorithm for this problem, we thought that quite likely that there must also be a polynomial-time deterministic algorithm since not many problems are known that have a polynomial-time randomized algorithm but no polynomial-time deterministic algorithm. [3] The randomized algorithm performs a forward analysis over the lattice of Herbrand equivalences. This pointed towards the existence of a deterministic forward analysis over

[3]Until recently, there were two interesting problems, polynomial identity testing, and primality testing, for which a polynomial-time randomized algorithm was known but no polynomial-time deterministic algorithm was known. Recently, primality testing was shown to be have a polynomial-time deterministic algorithm [AKS02].

the same lattice. We first guessed that fixed-point computation was not really the problem, as opposed to what was claimed in one of the earlier attempts [RKS99]; this even led us to uncover a bug in the completeness claim in the corresponding paper. The challenge that we were facing was in computing the join of two sets of Herbrand equivalences. The size of the result of a single join operation is, in the worst case, product of the sizes of the input sets, and a naive analysis means that the result after t joins may be exponential in t. We tried hard to prove better bounds for an amortized analysis only to come across a concrete example that in fact exhibited the worst case. (This example is described in [GN04b].) We then turned our attention to the randomized algorithm to see how it was representing such exponentially sized equivalences. The randomized algorithm represents equivalences of expressions of bounded depth, which is sufficient to identify equal sub-expressions in a program. This gave us the idea of performing an approximate join operation (in the deterministic algorithm) that maintains equivalences of bounded size.

The deterministic algorithm takes an integral parameter s_e (which denotes the maximum size of the expressions whose equivalences we are interested in) and discovers all equivalences of the form $e_1 = e_2$ if $size(e_1) \leq s_e$ and $size(e_2) \leq s_e$. Here, $size(e)$ denotes the number of function applications of the binary uninterpreted function F in the DAG representation of expression e. The algorithm uses a data structure called *Strong Equivalence DAG* (described in Section 3.3.1) to represent the set of equivalences at any program point. It updates the data structure across each flowchart node using the transfer functions described in Section 3.3.2.

3.3.1 Strong Equivalence DAG

The algorithm represents the set of equivalences at any program point by a data structure that we call *Strong Equivalence DAG* (SED). An SED is similar to a value graph [Muc00]. It is a labeled directed acyclic graph whose nodes η can be represented by tuples $\langle V, h\rangle$ where V is a (possibly empty) set of program variables labeling the node, and h represents the type of node. The type h is either $\top$ indicating that the node has no successors, or $F(\eta_1, \eta_2)$ indicating that the node has two ordered successors η_1 and η_2.

In any SED G, for every variable x, there is exactly one node $\langle V, h\rangle$, denoted by $Node_G(x)$, such that $x \in V$. For every type h that is not $\top$, there is at most one node with that type. For any SED node η, we use the notation $Vars(\eta)$ to denote the set of

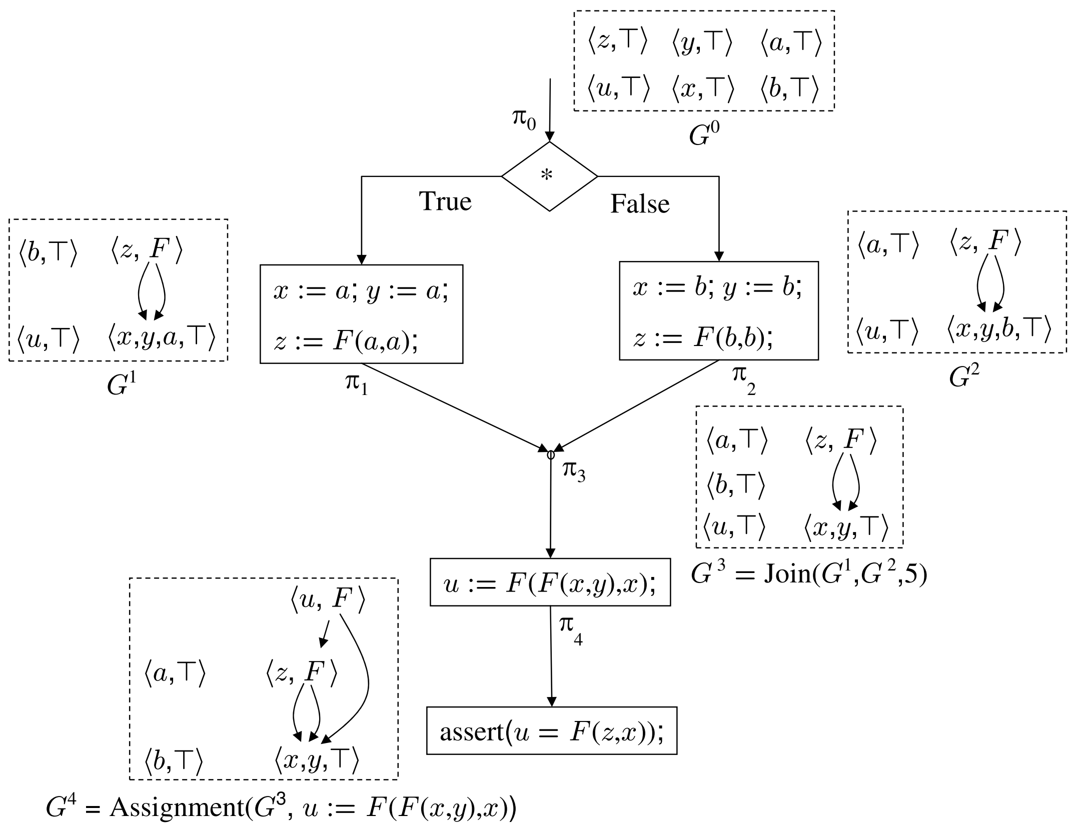


Figure 3.5: Computing Herbrand equivalences using abstract interpretation. G_i, shown in dotted box, represents the SED at program point π_i.

variables labeling node η, and $Type(\eta)$ to denote the type of node η. Every node η in an SED represents the following set of terms $Terms(\eta)$, which are all known to be equal.

$$\begin{aligned} Terms(\langle V, \top\rangle) &= V \\ Terms(\langle V, F(\eta_1,\eta_2)\rangle) &= V \cup \{F(e_1,e_2) \parallel e_1 \in Terms(\eta_1), e_2 \in Terms(\eta_2)\} \end{aligned}$$

We use the notation $G \models e_1 = e_2$ to denote that G implies the equivalence $e_1 = e_2$. The judgment $G \models e_1 = e_2$ is deduced as follows.

$$G \models F(e_1,e_2) = F(e_1',e_2') \text{ iff } G \models e_1 = e_1' \text{ and } G \models e_2 = e_2'$$
$$G \models x = e \text{ iff } e \in Terms(Node_G(x))$$

In figures showing SEDs, we omit the set delimiters "{" and "}", and represent a node $\langle\{x_1,..,x_j\},h\rangle$ as $\langle x_1,..,x_j,h\rangle$ for simplicity. Figure 3.5 shows a program and the SEDs computed by our algorithm at various points. As an example, note that $\mathit{Terms}(\mathit{Node}_{G_4}(u)) = \{u\} \cup \{F(z,\alpha) \parallel \alpha \in \{x,y\}\} \cup \{F(F(\alpha_1,\alpha_2),\alpha_3) \parallel \alpha_1,\alpha_2,\alpha_3 \in \{x,y\}\}$, and hence $G_4 \models u = F(z,x)$. Note that an SED represents compactly a possibly exponential number of equal terms.

For notational convenience, we extend the definition of an SED to also include an undefined SED, denoted by $\perp$. We also say that $\perp \models e_1 = e_2$ for all expressions e_1 and e_2.

3.3.2 Basic Algorithm

The algorithm computes the SEDs (which represent Herbrand equivalences) at each program point by performing a forward analysis on the flowchart nodes. The algorithm uses the following transfer functions to compute the SEDs across the flowchart nodes shown in Figure 3.4 (on page 54) until a fixed point is reached. It follows from Theorem 11 that a fixed point for a loop is reached in at most $k_{\mathtt{u}} + 1$ iterations.

Initialization: At procedure entry, the algorithm starts with the following initial SED G, which implies only trivial equivalences. The SEDs at other program points is initialized to be $\perp$.

$$G = \{\langle \{x\}, \top \rangle \parallel x \text{ is a program variable}\}$$

Assignment Node: See Figure 3.4 (a).
If the SED G' before the assignment node is $\perp$, then the SED G after the assignment node is also $\perp$. Otherwise, the SED G after an assignment node $x := e$ is obtained from the SED G' before the assignment node using the following `Assignment` operation. SED G^4 in Figure 3.5 shows an example of the `Assignment` operation.

```
Assignment($G', x := e$) =
    $G := G'$;
    let $\langle V_1, h_1 \rangle$ = GetNode($G, e$) in
    let $\langle V_2, h_2 \rangle = \mathit{Node}_G(x)$ in
    ReplaceVars($G, \langle V_1, h_1 \rangle, V_1 \cup \{x\}$);
    ReplaceVars($G, \langle V_2, h_2 \rangle, V_2 - \{x\}$);
    return $G$;
```

```
GetNode($G, e$) =
  match $e$ with
    $y$:  return $Node_G(y)$;
    $F(e_1, e_2)$:  let $\eta_1$ = GetNode($G, e_1$) and $\eta_2$ = GetNode($G, e_2$) in
                if $\langle V, F(\eta_1, \eta_2)\rangle \in G$ for some $V$ then return $\langle V, F(\eta_1, \eta_2)\rangle$;
                else $G := G \cup \langle \emptyset, F(\eta_1, \eta_2)\rangle$; return $\langle \emptyset, F(\eta_1, \eta_2)\rangle$;
```

GetNode(G, e) returns a node η such that $e \in \mathit{Terms}(\eta)$ (and in the process possibly extends G) in $O(\mathit{size}(e))$ time. ReplaceVars(G, η, V) replaces the set of variables in node η by V (in place) in SED G. Lines 4 and 5 in Assignment operation move variable x to the node GetNode(G, e) to reflect the equivalence $x = e$. Hence, the following lemma holds.

Lemma 10 [Soundness and Completeness of Assignment Operation] *Let* $G =$ Assignment$(G', x := e)$. *Let* e_1 *and* e_2 *be two expressions. Let* $e_1' = e_1[e/x]$ *and* $e_2' = e_2[e/x]$. *Then,* $G \models e_1 = e_2$ *iff* $G' \models e_1' = e_2'$.

Non-deterministic Assignment Node: See Figure 3.4 (b).
If the SED G' before the non-deterministic assignment node is $\perp$, then the SED G after the assignment node is also $\perp$. Otherwise, the SED G after a non-deterministic assignment node $x :=?$ is simply obtained from the SED G' before the non-deterministic assignment node by removing variable x from $Node_{G'}(x)$, and creating a new node $\langle \{x\}, \top\rangle$.

```
Non-det-Assignment($G', x :=?$) =
  $G := G'$;
  let $\langle V, h\rangle = Node_G(x)$ in
  ReplaceVars($G, \langle V, h\rangle, V - \{x\}$);
  $G := G \cup \{\langle \{x\}, \top\rangle\}$;
  return $G$;
```

Non-deterministic Conditional Node: See Figure 3.4 (c).
The SEDs G^1 and G^2 on the two branches of the non-deterministic conditional node are simply a copy of the SED G before the non-deterministic conditional node.

Join Node: See Figure 3.4 (d).
If at least one of the SEDs G^1 or G^2 before the join node is $\perp$, then the SED after the join

node is assigned the other SED. Otherwise, the SED G after the join node is computed using the following `Join` operation. `Join` takes two SEDs G^1 and G^2, and a positive integer s'_e as input, and returns an SED G that represents all equivalences $e_1 = e_2$ such that both G^1 and G^2 imply $e_1 = e_2$ and both $size(e_1)$ and $size(e_2)$ are at most s'_e. In order to discover all equivalences among expressions of size at most s_e in the program, we need to choose $s'_e = s_e + s_m(k_{\mathtt{u}} + 1)$ (for reasons explained later in Section 3.3.3), where s_m is the maximum size of any expression computed by the program along any acyclic path without taking any back-edge. Figure 3.5 shows an example of the `Join` operation.

For any SED G, let $\prec_G$ denote a partial order on program variables such that $x \prec_G y$ if y depends on x, or more precisely, if $G \models y = F(e_1, e_2)$ and $x \in Vars(F(e_1, e_2))$.

```
Join($G^1$,$G^2$,$s'_e$) =
    for all nodes $\eta_1 \in G^1$ and $\eta_2 \in G^2$ do
        $memoize[\eta_1, \eta_2] := \bot$;
    $G := \emptyset$;
    for each program variable $x$ in the order $\prec_{G^1}$ do
        $counter := s'_e$;
        Intersect($Node_{G^1}(x), Node_{G^2}(x)$);
    return $G$;
```

```
Intersect($\langle V_1, h_1\rangle$,$\langle V_2, h_2\rangle$) =
    let $\eta = memoize(\langle V_1, h_1\rangle, \langle V_2, h_2\rangle)$ in
    if $\eta \neq \bot$ then return $\eta$;
    let $h =$ if $counter > 0$ and $h_1 \equiv F(a_1, b_1)$ and $h_2 \equiv F(a_2, b_2)$ then
                $counter := counter - 1$;
                let $a$ = Intersect($a_1$,$a_2$) in
                let $b$ = Intersect($b_1$,$b_2$) in
                if $(a \neq \langle\emptyset, \top\rangle)$ and $(b \neq \langle\emptyset, \top\rangle)$ then $F(a, b)$ else $\top$
             else $\top$ in
    let $V = V_1 \cap V_2$ in
    if $V \neq \emptyset$ or $h \neq \top$ then $G := G \cup \{\langle V, h\rangle\}$;
    $memoize[\langle V_1, h_1\rangle, \langle V_2, h_2\rangle] := \langle V, h\rangle$;
    return $\langle V, h\rangle$;
```

It is important for correctness of the `Join` operation that calls to the `Intersect` function are memoized, as done explicitly in the above pseudo code, since otherwise the *counter* variable will be decremented incorrectly. The use of *counter* variable ensures that the call to `Intersect` function in `Join` terminates in $O(s'_e)$ time. The following property of `Intersect` function is required to prove the correctness of the `Join` operation (Lemma 11).

Property 4 *Let* $\eta_1 = \langle V_1, h_1 \rangle$ *and* $\eta_2 = \langle V_2, h_2 \rangle$ *be any nodes in SEDs* G^1 *and* G^2 *respectively. Let* $\eta = \langle V, h \rangle = \texttt{Intersect}(\eta_1, \eta_2)$. *Suppose that* $\eta \neq \langle \emptyset, \top \rangle$*; hence the function* $\texttt{Intersect}(\eta_1, \eta_2)$ *adds the node* η *to* G. *Let* α *be the value of the counter variable when* $\texttt{Intersect}(\eta_1, \eta_2)$ *is first called. Then,*

A1. $\mathit{Terms}(\eta) \subseteq \mathit{Terms}(\eta_1) \cap \mathit{Terms}(\eta_2)$.

A2. $\mathit{Terms}(\eta) \supseteq \{e \parallel e \in \mathit{Terms}(\eta_1), e \in \mathit{Terms}(\eta_2), \mathit{size}(e) \leq \alpha\}$.

The proof of Property 4 is by induction on sum of height of nodes η_1 and η_2 in G^1 and G^2 respectively. Claim A1 is easy since $h = F(...)$ only if both h_1 and h_2 are $F(...)$ (Line 7), and $V = V_1 \cap V_2$ (Line 9). The proof of claim A2 relies on bottom-up processing of one of the SEDs, and memoization. Let e' be one of the *smallest* expressions (in terms of *size*) such that $e' \in \mathit{Terms}(\eta_1) \cap \mathit{Terms}(\eta_2)$. If e' is not a variable, then for any variable $y \in \mathit{Vars}(e')$, the call $\texttt{Intersect}(\mathit{Node}_{G^1}(y), \mathit{Node}_{G^2}(y))$ has already finished. The crucial observation now is that if $\mathit{size}(e') \leq \alpha$, then the set of recursive calls to `Intersect` are in 1-1 correspondence with the nodes of expression e', and $e' \in \mathit{Terms}(\eta)$.

Lemma 11 [Soundness and Completeness of Join Operation] *Let* $G = \texttt{Join}(G^1, G^2, s)$. *If* $G \models e_1 = e_2$, *then* $G^1 \models e_1 = e_2$ *and* $G^2 \models e_1 = e_2$. *If* $G^1 \models e_1 = e_2$, $G^2 \models e_1 = e_2$ *and* $\mathit{size}(e_1) \leq s$, $\mathit{size}(e_2) \leq s$, *then* $G \models e_1 = e_2$.

The proof of Lemma 11 follows from Property 4 and definition of $\models$.

Verifying or Discovering Equal Program Sub-expressions

After a fixed point has been reached, the SEDs computed by the algorithm at various program points can be used to decide equivalences of expressions at those points. Let G be the SED computed by the algorithm at a program point π after fixed-point computation. If $G = \perp$, the program point π is unreachable. Else, the algorithm declares expressions e_1 and e_2 to be equal at program point π iff $G \models e_1 = e_2$.

Optimization

Maintaining an SED explicitly at each program point is expensive (in terms of time and space complexity) and redundant. We can instead maintain one global SED G for the SSA version of the program (similar to the optimization discussed in Section 3.2.1). This requires small modifications to the `Assignment`, `Non-det-Assignment`, and the `Join` operations defined earlier.

The `Assignment` and `Non-det-Assignment` operations remain essentially the same except that they do not create a new SED. More concretely, lines 1 and 6 in the description of the `Assignment` operation, while lines 1 and 5 in the description of the `Non-det-Assignment` operation are removed. Also, all occurrences of G' in those operations are replaced by G.

The `Join` operation involves processing all phi-assignments at the corresponding join point π. Let x_π denote the renamed version of variable x at join point π after the SSA conversion. For each phi-assignment $x_\pi = \phi(x_{\pi_1}, x_{\pi_2})$ at join point π, the `Join` operation involves executing `Intersect`($Node_G(x_{\pi_1})$,$Node_G(x_{\pi_2})$) as shown below:

```
Join($G$,$\pi$,$s'_e$) =
1    for all nodes $\eta_1, \eta_2 \in G$ do
2        $memoize[\eta_1, \eta_2] := \bot$;
3    for each phi-assignment $x_\pi = \phi(x_{\pi_1}, x_{\pi_2})$ in $G$ do
4        let $\langle V, h\rangle = Node_G(x_\pi)$ in
5        $G := G - \langle V, h\rangle$;
6        if $V - \{x_\pi\} \neq \emptyset$ then $G := G \cup \{\langle V - \{x_\pi\}, h\rangle\}$;
7    for each phi-assignment $x_\pi = \phi(x_{\pi_1}, x_{\pi_2})$ in $G$ in the order $\prec_G$ do
8        $counter := s'_e$;
9        Intersect($Node_G(x_{\pi_1}), Node_G(x_{\pi_2})$);
```

The order $\prec_G$ in the loop in Line 7 refers to the ordering of phi-variables x_π. We can avoid the costly step of initializing the *memoize* data structure by implementing it as a hash table rather than an array. The `Intersect` routine stays same as before except the definition of V in line 9 is refined to $\{target(y) \parallel y \in V_1\} \cap \{target(y) \parallel y \in V_2\}$ instead of $V_1 \cap V_2$. The function $target(y)$ returns the target variable of the phi-assignment at join point π in which the variable y occurs; if no such phi-assignment exists, then $target(y)$ returns y. For example, if there is a phi-assignment $y_\pi = \phi(y_{\pi_1}, y_{\pi_2})$, then $target(y_{\pi_1}) = target(y_{\pi_2}) = y_\pi$.

The `Intersect` routine can be optimized to immediately return node η if the two inputs are the same node η.

For detecting whether a fixed point across a loop has been reached or not, there is no need to maintain old copies of the SEDs. It follows from the proof of the fixed point theorem (Lemma 9) that if a fixed point is not reached, then the number of independent variables will be different. Hence, in order to detect whether a fixed point for a loop has been reached or not, simply a count of the number of independent variables at the target of all back-edges for that loop suffices.

3.3.3 Correctness

Theorem 12 and Theorem 13 below imply that the algorithm computes all equivalences between program expressions of size at most s_e at each program point. This implies that if we choose s_e to be the number of occurrences of the uninterpreted function symbol F in the program, then the algorithm detects all equivalences among program sub-expressions.

Theorem 12 [Soundness Theorem] *Let G be the SED computed by the algorithm at some program point π after fixed-point computation. If $G \models e_1 = e_2$, then $e_1 = e_2$ holds at program point π.*

Theorem 12 can be easily proved by induction on the number of operations performed by the algorithm using soundness of the individual operations. Soundness of the operations performed for assignment and join nodes is stated by Lemma 10 and Lemma 11 respectively, while soundness of the operations for non-deterministic assignment and conditional nodes is trivial.

Theorem 13 [Completeness Theorem] *Let $e_1 = e_2$ be an equivalence that holds at a program point π such that size$(e_1) \leq s_e$ and size$(e_2) \leq s_e$. Let G be the SED computed by the algorithm at program point π after fixed-point computation. Then, $G \models e_1 = e_2$.*

The proof of Theorem 13 follows from an invariant maintained by the algorithm at each program point.

Lemma 12 *Let G be the SED computed by the algorithm at some program point π. Let T be the set of program paths (each from procedure entry to program point π) that have been analyzed by the algorithm immediately after computation of G. Let s be a bound on the size* *of the expressions (in terms of the number of occurrences of the function symbol F in the DAG representation) computed by the program along any path in T. Suppose $e_1 = e_2$ holds at program point π along all paths in T, size$(e_1) \leq s'_e - s$ and size$(e_2) \leq s'_e - s$. Then, $G \models e_1 = e_2$.*

Lemma 12 can be easily proved by induction on the number of operations performed by the algorithm using completeness of the individual operations. Completeness of the operations performed for assignment and join nodes is stated by Lemma 10 and Lemma 11 respectively, while completeness of the operations for non-deterministic assignment and conditionals nodes is trivial.

Theorem 11 (the fixed point theorem) requires the algorithm to execute each node at most $k_{\mathtt{u}} + 1$ times (assuming the standard worklist implementation [Muc00]). This implies that the integer s (in the statement of Lemma 12) at any program point after fixed-point computation is at most $s_m(k_{\mathtt{u}} + 1)$ (where s_m is the maximum size of any expression computed by the program along any acyclic path without taking any back-edge). Hence, choosing $s'_e = s_e + s_m(k_{\mathtt{u}} + 1)$ enables the algorithm to discover equivalences among expressions of size s_e. The proof of Theorem 13 now follows easily from Lemma 12.

3.3.4 Computational Complexity

We assume that the deterministic algorithm performs the optimization of maintaining one global SED for the SSA version of the program as discussed in Section 3.3.2. An assignment operation takes constant time. Processing each phi-assignment in a join operation takes time $O(s_m k_{\mathtt{u}})$, where s_m is the maximum size of any expression computed by the program along any acyclic path without taking any back-edge. It follows from Theorem 11 that each flowchart node is processed at most $k_{\mathtt{u}} + 1$ times. Hence, the total cost of all assignment and join operations is $O(n_{\mathtt{s}} s_m k_{\mathtt{u}}^2)$. In comparison, the complexity of the randomized algorithm presented earlier is slightly better: $O(n_{\mathtt{s}} d_m k_{\mathtt{u}}^2)$, where d_m is the maximum depth of any expression computed by the program along any acyclic path without taking any back-edge. However, the complexity analysis of the randomized algorithm appears to be a conservative one though, and may be the case that the randomized algorithm has a better computational complexity of $O(n_{\mathtt{s}} k_{\mathtt{u}} \log n_{\mathtt{s}})$ (as described in Section 3.2.4).

3.4 Related Work

There have been several proposed algorithms for discovering Herbrand equivalences. However, all these algorithms either take exponential time, or do not discover all equivalences among program sub-expressions (for the abstraction where program operators are modeled as uninterpreted functions, and conditionals are treated as non-deterministic).

Kildall's Algorithm: Kildall's algorithm [Kil73] performs an abstract interpretation over the lattice of sets of Herbrand equivalences. It represents the set of Herbrand equivalences at each program point by means of a structured partition.

The join operation for two structured partitions is defined to be their intersection. Kildall's algorithm is complete in the sense that if it terminates, then the structured partition at any program point reflects all Herbrand equivalences that are true at that point. However, the complexity of Kildall's algorithm is exponential. The number of elements in a partition, and the size of each element in a partition can all be exponential in the number of join operations performed.

Alpern, Wegman and Zadeck's (AWZ) Algorithm: The AWZ algorithm [AWZ88] works on the value graph representation [Muc00] of a program that has been converted to SSA form. A value graph can be represented by a collection of nodes of the form $\langle V, h\rangle$ where V is a set of variables, and the type h is either $\top$ (indicating that the node has no successors), $F(\eta_1, \eta_2)$ or $\phi^j(\eta_1, \eta_2)$ (indicating that the node has two ordered successors η_1 and η_2). ϕ^j denotes the ϕ function associated with the j^{th} join point in the program. Our data structure SED can be regarded as a special form of a value graph which is acyclic and has no ϕ-type nodes. The main step in the AWZ algorithm is to use congruence partitioning to merge some nodes of the value graph.

The AWZ algorithm cannot discover all equivalences among program terms. This is because it treats ϕ functions as uninterpreted. The ϕ functions are an abstraction of the if-then-else operator wherein the conditional in the if-then-else expression is abstracted away, but the two possible values of the if-then-else expression are retained. The ϕ functions satisfy the axioms stated in Equation 3.6 and Equation 3.7.

Rüthing, Knoop and Steffen's (RKS) Algorithm: Like the AWZ algorithm, the RKS algorithm [RKS99] also works on the value graph representation of a program that has been converted to SSA form. It tries to capture the semantics of ϕ functions by applying the following rewrite rules, which are based on Equation 3.6 and Equation 3.7, to convert program expressions into some normal form.

$$\langle V, \phi^j(\eta, \eta)\rangle \text{ and } \eta \rightarrow \langle V \cup \mathit{Vars}(\eta), \mathit{Type}(\eta)\rangle \tag{3.11}$$

$$\langle V, \phi^j(\langle V_1, F(\eta_1, \eta_2)\rangle, \langle V_2, F(\eta_3, \eta_4)\rangle)\rangle \rightarrow \langle V, F(\langle \emptyset, \phi^j(\eta_1, \eta_3)\rangle, \langle \emptyset, \phi^j(\eta_2, \eta_4)\rangle)\rangle \tag{3.12}$$

Nodes on left side of the rewrite rules are replaced by the new node on right side of the rewrite rules, and incoming edges to nodes on left side of the rewrite rules are made to point to the new node. However, there is a precondition to applying the second rewriting rule.

$$\text{Precondition: } \forall \text{ nodes } \eta \in \mathit{succ}^*(\{\langle V_1, F(\eta_1, \eta_2)\rangle, \langle V_2, F(\eta_3, \eta_4)\rangle\}),\ \mathit{Vars}(\eta) \neq \emptyset$$

The RKS algorithm assumes that all assignments are of the form $x := F(y, z)$ to make sure that for all original nodes η in the value graph, $\mathit{Vars}(\eta) \neq \emptyset$. The notation $\mathit{succ}^*(M)$ denotes the transitive closure of the successor of all nodes in set M. (A node with type $F(\eta_1, \eta_2)$ or $\phi^j(\eta_1, \eta_2)$ has nodes η_1 and η_2 as its successors, while a node with type $\top$ has no successor.) This precondition is necessary in arguing termination for the above system of rewrite rules, and proving the polynomial complexity bound. The RKS algorithm alternately applies the AWZ algorithm and the above two rewrite rules until the value graph reaches a fixed point. Thus, the RKS algorithm discovers more equivalences than the AWZ algorithm.

The RKS algorithm cannot discover all equivalences even in acyclic programs. This is because the above precondition can prevent two equal expressions from reaching the same normal form. On the other hand lifting the precondition may result in the creation of an exponential number of new nodes, and an exponential number of applications of the rewrite rules.

The RKS algorithm has another problem, which the authors have identified. It fails to discover all equivalences in cyclic programs, even if the precondition described above is lifted. This is because the graph rewrite rules add a degree of pessimism to the iteration process. While congruence partitioning is optimistic, it relies on the result of the graph transformations, which are pessimistic since they are applied outside of the fixed-point iteration process.

Randomized algorithms have always been characterized by the following dichotomy: on one hand, they are natural and simple to describe and understand, sometimes almost childishly so. On the other hand, their analysis often requires deep mathematical techniques, that are also invariably, strikingly beautiful and elegant.

Devdatt P. Dubhashi, *Report on a Workshop on Randomized Algorithms.*

Chapter 4

Inter-procedural Analysis

The analyses described in the previous two chapters are intra-procedural analyses, i.e., they analyze each procedure in a program in isolation. More precise reasoning can be done for a program using a (context-sensitive) inter-procedural analysis, which involves analyzing all procedures in a program in presence of each other. This is because a procedure in a program may be called at several places in that program only with some specific input values. If something more is known about the input variables of a procedure, then it becomes possible to say more interesting things about the behavior of the procedure, as opposed to an intra-procedural analysis, which assumes that a procedure may be called with all possible values for the input variables.

One way to do inter-procedural analysis is to do procedure-inlining followed by an intra-procedural analysis. There are two potential problems with this approach. First, in presence of recursive procedures, procedure-inlining may not be possible. Second, even if there are no recursive procedures, procedure-inlining may result in an exponential blow-up of the program. For example, if procedure P_1 calls procedure P_2 two times, which in turn calls procedure P_3 two times, then procedure inlining will result in 4 copies of procedure P_3 inside procedure P_1. In general, leaf procedures can be replicated an exponential number of times.

A more standard technique to do inter-procedural analysis is by means of computing procedure summaries. Each procedure is analyzed once (or a few times in case of recursive procedures) to build its summary. A procedure summary can be thought of as some succinct representation of the behavior of the procedure that is also parametrized by any information about its input variables. In general, there is no automatic recipe to construct these procedure summaries, and abstraction specific techniques are required.

In this chapter, we describe how to efficiently extend random interpretation based intra-procedural analyses to an inter-procedural setting by means of computing random procedure summaries [GN05]. For this purpose we first describe a unified framework for random interpretation that generalizes previous randomized intra-procedural analyses, and also extends naturally to efficient inter-procedural analyses. We then discuss the two key ideas required to extend an intra-procedural random interpreter to an inter-procedural setting. In Section 4.3.1, we use the unified framework and the key ideas to describe a generic inter-procedural random interpreter.

4.1 Framework for Random Interpretation

An intra-procedural random interpreter executes a program on random inputs in a non-standard manner as described in previous chapters. It computes a state ρ at each program point by performing a forward analysis on the program. A state is a mapping from program variables to values v that are polynomials over the field $\mathbb{F}_p$. These polynomials may simply be elements of $\mathbb{F}_p$ (as in Chapter 2 for linear arithmetic analysis), vectors of elements from $\mathbb{F}_p$ [1] (as in Chapter 3 for uninterpreted functions analysis), or linear functions of the program's input variables (as in this chapter for inter-procedural analysis).

The random interpreter copies the state before a non-deterministic conditional node on its two branches. It computes the state after a join node by performing an affine join of the states before the join node. For computing the state after an assignment node, it uses the abstraction-specific `SEval` function as described below.

[1] A vector $(v_1, \ldots, v_\ell)$ can be represented by the polynomial $\sum_{i=1}^{\ell} z_i v_i$, where $z_1, \ldots, z_\ell$ are some fresh variables.

4.1.1 `SEval` Function

The random interpreter processes an assignment node $x := e$ by updating the value of variable x to the value of expression e in the state before the assignment node. Expressions are evaluated using the `Eval` function, which depends on the underlying abstract domain of the analysis. The `Eval` function takes an expression e and a state ρ and computes some value v. The `Eval` function plays the same role as an abstract interpreter's transfer function for an assignment node. `Eval` is defined in terms of a symbolic function `SEval` that translates an expression into a polynomial over the field $\mathbb{F}_p$. This polynomial is linear in program variables, and may contain random variables as well, which stand for random field values chosen during the analysis. (The `SEval` function for linear arithmetic has no random variables, while the `SEval` function for uninterpreted functions uses random variables.) `Eval`(e, ρ) is computed by replacing program variables in `SEval`(e) with their values in state ρ, replacing the random variables with the random values that have chosen for them, and then evaluating the result over the field $\mathbb{F}_p$. (The random values r_j are chosen once for the random variables y_j, and the same value r_j is used for all occurrences of the random variable y_j.) Following are examples of two `Eval` functions that we have seen in the earlier chapters.

`SEval` function for Linear Arithmetic The random interpretation for linear arithmetic is described in Chapter 2. The following language describes the expressions in this abstract domain. Here x refers to a variable and c refers to an arithmetic constant.

$$e ::= x \quad | \quad e_1 \pm e_2 \quad | \quad c \times e$$

The `SEval` function for this abstraction simply translates the linear arithmetic operations to the corresponding field operations. In essence, `Eval` simply evaluates the linear expression over the field $\mathbb{F}_p$.

$$\texttt{SEval}(e) = e$$

`SEval` function for Unary Uninterpreted Functions The random interpretation for uninterpreted functions is described in Chapter 3. We show here a simpler `SEval` function, for the case of unary uninterpreted functions. The following language describes the expressions in this abstract domain. Here x refers to a variable and F refers to a unary uninterpreted function.

$$e ::= x \quad | \quad F(e)$$

The `SEval` function for this abstraction is as follows.

$$\texttt{SEval}(x) = x$$
$$\texttt{SEval}(F(e)) = r_1 \times \texttt{SEval}(e) + r_2$$

Here r_1 and r_2 refer to random variables, unique for each unary uninterpreted function F. Note that in this case, `SEval` produces polynomials that have degree more than 1, although still linear in the program variables.

The `SEval` function corresponding to the `Eval` function for binary uninterpreted functions as described in Chapter 3 evaluates expressions to vectors (of polynomials). A vector $(v_1, \ldots, v_\ell)$ can however be represented as the polynomial $z_1 v_1 + \ldots + z_\ell v_\ell$ where $z_1, \ldots, z_\ell$ are fresh variables that do not occur in the program.

Properties of `SEval` Function

The `SEval` function should have the following properties. Let x be any variable and e_1 and e_2 be any expressions. Then,

B1. Soundness: The `SEval` function should not introduce any new equivalences.

$$\texttt{SEval}(e_1) = \texttt{SEval}(e_2) \Rightarrow e_1 = e_2$$

Note that the first equality is over polynomials, while the second equality is in the analysis domain.

B2. Completeness: The `SEval` function should preserve all equivalences.

$$e_1 = e_2 \Rightarrow \texttt{SEval}(e_1) = \texttt{SEval}(e_2)$$

B3. Referential transparency:

$$\texttt{SEval}(e_1[e_2/x]) = \texttt{SEval}(e_1)[\texttt{SEval}(e_2)/x]$$

This property (along with properties B1 and B2) is needed to prove the correctness of the action of the random interpreter for an assignment node $x := e$. As mentioned earlier, the random interpreter computes the state after an assignment node by updating the value of variable x to the value of the polynomial $\texttt{SEval}(e)$ in the state before the assignment node.

B4. Linearity: The `SEval` function should be a polynomial that is linear in program variables. This property is needed to prove the completeness of the random interpreter across join nodes, where it uses the affine join operator to merge program states.

Properties B1 and B3 are necessary for proving the probabilistic soundness of the random interpreter. Property B2 or property B4 need not be satisfied if completeness is not an issue. This may happen when the underlying abstraction is difficult to reason about, yet one is interested in a (probabilistically) sound and partially complete reasoning for that abstraction. For example, the following `SEval` function for "bitwise or operator" ($||$) satisfies all the above properties except property B2.

$$\texttt{SEval}(e_1||e_2) = \texttt{SEval}(e_1) + \texttt{SEval}(e_2)$$

This `SEval` function models commutativity and associativity of the $||$ operator. However, it does not model the fact that $x||x = x$. In Chapter 5, we will see an example of an `SEval` function that satisfies all the above properties except property B4. In this chapter, we assume that the `SEval` function satisfies all the properties mentioned above. However, the results of our chapter can also be extended to prove relative completeness of the random interpreter if the `SEval` function does not satisfy property B2 or property B4.

Also, note that the `SEval` function for linear arithmetic as described above has the soundness property for linear arithmetic over the prime field $\mathbb{F}_p$. The problem of reasoning about linear arithmetic over rationals can be reduced to reasoning about linear arithmetic over $\mathbb{F}_p$, where p is chosen randomly. This is a randomized reduction with some error probability, and is discussed further in Section 4.4.1.

4.2 Key Ideas

Inter-procedural random interpretation is based on the standard summary-based approach to inter-procedural analysis. Procedure summaries are computed in the first phase, and actual results are computed in the second phase. The real challenge is in computing context-sensitive summaries, i.e., summaries that can be instantiated with any context to

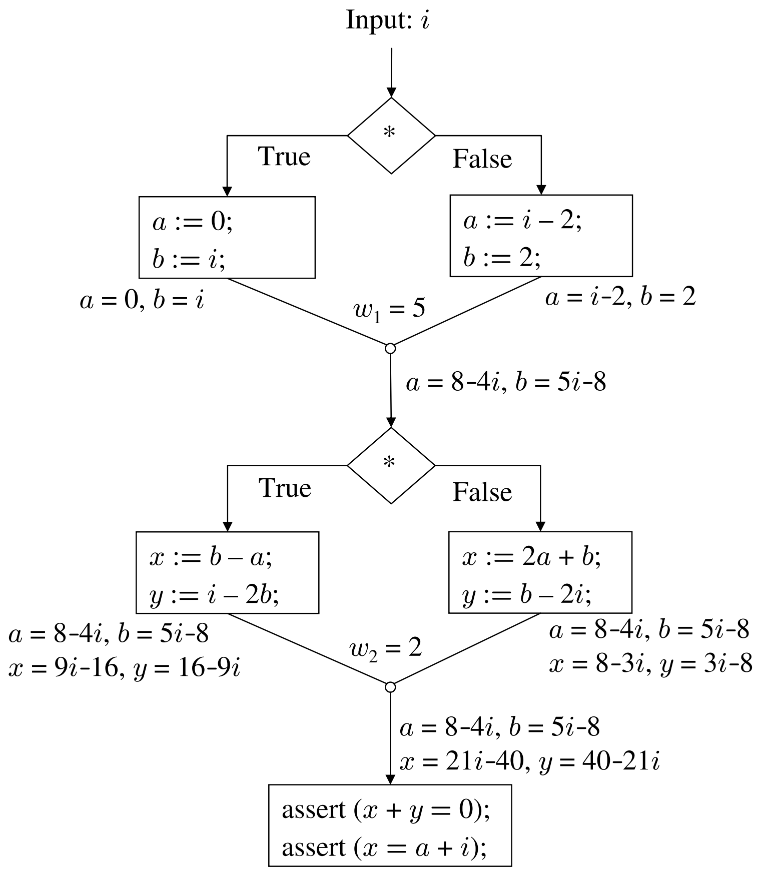

Input: i
$*$
True
False
$a := 0;$
$b := i;$
$a := i - 2;$
$b := 2;$
$a = 0, b = i$
$w_1 = 5$
$a = i\text{-}2, b = 2$
$a = 8\text{-}4i, b = 5i\text{-}8$
$*$
True
False
$x := b - a;$
$y := i - 2b;$
$x := 2a + b;$
$y := b - 2i;$
$a = 8\text{-}4i, b = 5i\text{-}8$
$x = 9i\text{-}16, y = 16\text{-}9i$
$w_2 = 2$
$a = 8\text{-}4i, b = 5i\text{-}8$
$x = 8\text{-}3i, y = 3i\text{-}8$
$a = 8\text{-}4i, b = 5i\text{-}8$
$x = 21i\text{-}40, y = 40\text{-}21i$
assert $(x + y = 0)$;
assert $(x = a + i)$;

Figure 4.1: Illustration of random symbolic interpretation for inter-procedural analysis on the program shown in Figure 2.2. Note that the second assertion is true in the context $i = 2$, and the random symbolic program state at the end of the program satisfies it in that context.

yield the most precise behavior of the procedures under that context. A context for a procedure refers to any relevant information regarding the values that the input variables of that procedure can take.

In this section, we briefly explain the two main ideas behind the summary computation technique that can be used to perform a precise inter-procedural analysis using the `SEval` function of a precise intra-procedural random interpreter.

4.2.1 Random Symbolic Run

Intra-procedural random interpretation involves interpreting a program using random values for the input variables. The state at the end of the procedure can be used as a summary for that procedure. However, such a summary will not be context-sensitive. For example, consider the procedure shown in Figure 2.2 (on page 13). The second assertion at the end of the procedure is true in the context $i = 2$, but this conditional fact is not captured by the random state at the end of the procedure.

Observe that in order to make the random interpretation scheme context-sensitive, we can simply delay choosing random values for the input variables. Instead of using states that map variables to field values, we use states that map variables to linear functions of input variables. This allows the flexibility to replace the input variables later depending on the context. However, we continue to choose random weights at join points and perform a random affine join operation.

As an example, consider again the procedure from before, shown now in Figure 4.1. Note that the random symbolic state at the end of the procedure (correctly) does not satisfy the second assertion. However, in a context where $i = 2$, the state does satisfy $x = a + i$ since x evaluates to 2 and a to 0. This scheme of computing partly symbolic summaries is surprisingly effective and guarantees context-sensitivity, i.e., it entails all valid equivalences in all contexts.

4.2.2 Multiple Runs

Consider the program shown in Figure 4.2. The first assertion in procedure B is true. However, the second assertion is false because the non-deterministic conditional in procedure A can branch differently in the two calls to procedure A, even with the same input. If we use the same random symbolic run for procedure A at different call sites in procedure B, then we incorrectly conclude that the second assertion holds. This happens because use of the same run at different call sites assumes that the non-deterministic conditionals in the called procedure are resolved in the same manner in different calls. This problem can be avoided if a fresh or independent run is used at each call point. By *fresh run*, we mean a run computed with a fresh choice of random weights at the join points.

One approach to generate a fresh run for each call site is to compute summaries that are parametrized by weight variables (i.e., instead of using random weights for performing the affine join operation, we use symbolic weight variables). Then, for each call site, we can instantiate this summary with a fresh set of random weights for the weight variables. The problem with this approach is that the symbolic coefficients of linear functions of input

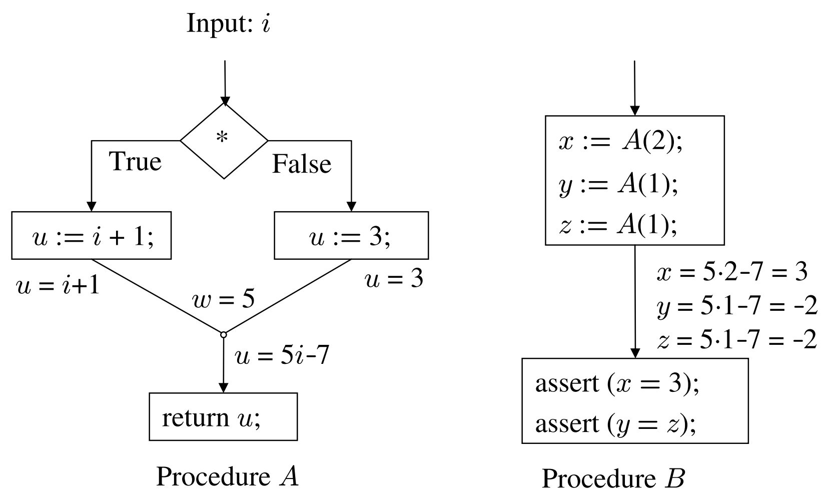


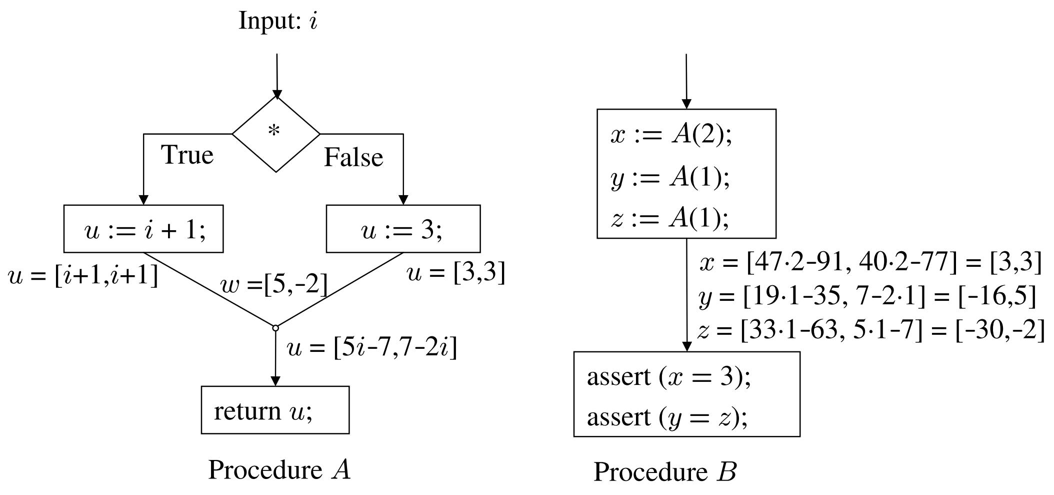
Figure 4.2: A program that demonstrates unsoundness of a single random symbolic run. Note that the first assertion at the end of procedure B is true, while the second assertion is not true since procedure A may take different branches in different runs.

variables, which are assigned to procedure variables (in states computed by the random interpreter) may have an exponential-size representation.

Another approach to generate m fresh runs for any procedure P is to execute m times the random interpretation scheme for procedure P, each time with a fresh set of random weights. However, this may require computing an exponential number of runs for other procedures. For example, consider a program in which each procedure P_i calls procedure P_{i+1} two times. To generate a run for P_0, we need 2 fresh runs for P_1, which are obtained using 4 fresh runs for P_2, and so on.

The approach that we use is to generate the equivalent of t fresh runs for any procedure P from t fresh runs of each of the procedures that P calls (for some parameter t that depends on the underlying abstraction). This approach relies on the fact that a random affine combination (i.e., a random weighted combination with sum of the weights being 1) of t runs of a procedure yields the equivalent of a fresh run for that procedure. For an informal geometric intuition, note that we can obtain any number of fresh points in a 2-dimensional plane by taking independent random affine combinations of three points that span the plane.

In Figure 4.3, we revisit the program shown in Figure 4.2 and illustrate this random interpretation technique of using a fresh run of a procedure at each call site. Note that we



Run 1: $u = \phi_7(5i\text{-}7, 7\text{-}2i) = 47i\text{-}91$
Run 1′: $u = \phi_6(5i\text{-}7, 7\text{-}2i) = 40i\text{-}77$
Run 2: $u = \phi_3(5i\text{-}7, 7\text{-}2i) = 19i\text{-}35$
Run 2′: $u = \phi_0(5i\text{-}7, 7\text{-}2i) = 7\text{-}2i$
Run 3: $u = \phi_5(5i\text{-}7, 7\text{-}2i) = 33i\text{-}63$
Run 3′: $u = \phi_1(5i\text{-}7, 7\text{-}2i) = 5i\text{-}7$
Fresh Runs for Procedure A

Figure 4.3: Illustration of multiple random symbolic runs for inter-procedural analysis on the program also shown in Figure 4.2. In this example, 2 random symbolic runs are computed for each procedure, and are further used to generate a fresh random symbolic run for every call to that procedure. Run j and Run j' are used at the j^{th} call site of procedure A while computing the two runs for procedure B. Note that this technique is able to correctly validate the first assertion and falsify the second one.

have chosen $t = 2$. The t runs of the procedure are shown in parallel by assigning a tuple of t values to each variable in the program. Note that procedure B calls procedure A three times. Hence, to compute 2 fresh runs for procedure B, we need to generate 6 fresh runs for procedure A. The figure shows generation of 6 fresh runs (Runs 1,1′,2,2′,3, and 3′) from the 2 runs for procedure A. The first call to procedure A uses the first two (Runs 1 and 1′) of these 6 runs, and so on. Note that the resulting program states at the end of procedure B satisfy the first assertion, but not the second assertion thereby correctly invalidating it.

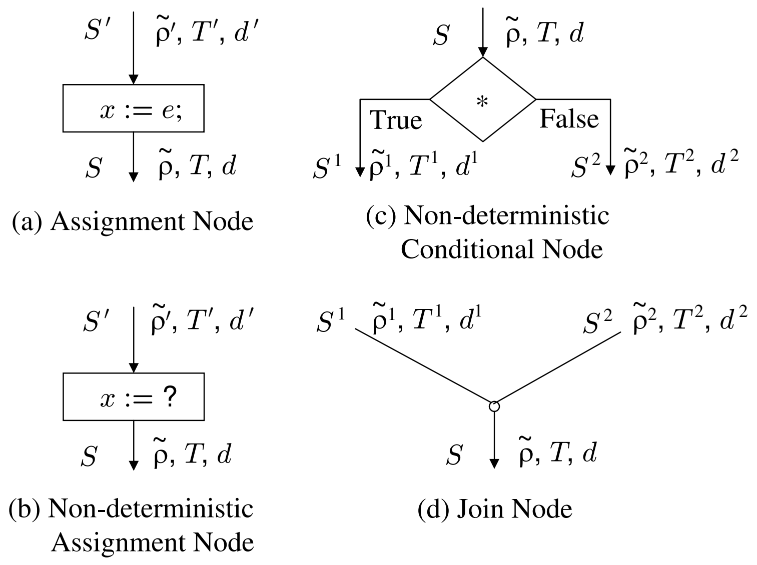

S' $\tilde{\rho}', T', d'$
$x := e;$
S $\tilde{\rho}, T, d$
(a) Assignment Node
S $\tilde{\rho}, T, d$
$*$
True
False
S^1 $\tilde{\rho}^1, T^1, d^1$
S^2 $\tilde{\rho}^2, T^2, d^2$
(c) Non-deterministic Conditional Node
S' $\tilde{\rho}', T', d'$
Call P$'$
S $\tilde{\rho}, T, d$
(e) Procedure Call
S' $\tilde{\rho}', T', d'$
$x :=$?
S $\tilde{\rho}, T, d$
(b) Non-deterministic Assignment Node
S^1 $\tilde{\rho}^1, T^1, d^1$
S^2 $\tilde{\rho}^2, T^2, d^2$
S $\tilde{\rho}, T, d$
(d) Join Node

Figure 4.4: Flowchart nodes considered in inter-procedural analysis.

4.3 The Random Interpreter

We now describe the precise inter-procedural random interpretation for any abstraction that is equipped with an `SEval` function that has the properties discussed in Section 4.1.1.

We assume that each procedure has been abstracted using the flowchart nodes shown in Figure 4.4. A procedure call node is simply denoted by the name of the procedure P' that is being called. For simplicity, we assume that the inputs and outputs of a procedure being called are passed as global variables.

Apart from the notation established in Section 2.2, we use the following notation related to our program model.

- $k_{\tt i}$: Maximum number of input variables for any procedure.
- $k_{\tt o}$: Maximum number of output variables for any procedure.
- $n_{\tt pp}$: Maximum number of procedure call nodes in any procedure.
- $n_{\tt p}$: Number of procedure call nodes.

The set of input variables of a procedure P includes the set of all global variables read by procedure P directly as well as the set of input variables for any procedure P' called by

P. Similarly for the set of output variables of a procedure P. Since we consider procedure calls nodes in this abstraction, $n = n_{\mathtt{a}} + 2n_{\mathtt{c}} + n_{\mathtt{p}}$ (instead of being just $n_{\mathtt{a}} + 2n_{\mathtt{c}}$, as in Section 2.2).

4.3.1 Basic Algorithm

The inter-procedural random interpreter performs a standard two-phase computation. The first phase, or the bottom-up phase, computes procedure summaries by starting with leaf procedures. The second phase, or top-down phase, computes the actual results of the analysis at each program point by using the summaries computed in the first phase. In presence of loops in the call graph and inside procedures, both phases require fixed-point computation, which we address in Section 4.3.3.

The random interpreter starts by choosing random elements $r_j \in \mathbb{F}_p$ for any random variables y_j that are used in the `SEval` function. Every occurrence of variable y_j is replaced by the same random choice r_j when evaluating any expression using the `Eval` function. We use the notation $\mathtt{SEval}'(e)$ to denote the polynomial obtained from $\mathtt{SEval}(e)$ by replacing all occurrences of the random variables y_j by the random elements r_j that have been chosen (globally) for them. The prime p is chosen to ensure that the error probability of the random interpreter (which is a function of p among other parameters, as described in Theorem 14) is small. A 32-bit prime is usually sufficient in practice. We now describe the two-phase computation performed by the random interpreter.

Phase 1

A summary for a procedure P, denoted by Y_P, is either $\perp$ (denoting that the procedure has not yet been analyzed, or on all paths it transitively calls procedures that have not yet been analyzed), or is a collection of t runs $\{Y_{P,i}\}_{i=1}^{t}$. A run of procedure P is a mapping from output variables of procedure P to random symbolic values, which are linear expressions in terms of the input variables of procedure P. The number of runs t should be greater than $k_{\mathtt{v}} + 2k_{\mathtt{i}}$ for probabilistic soundness, as predicted by our theoretical estimates. However, experiments (discussed in Section 4.6) suggest that a smaller value of t does not yield any error in practice.

To compute a procedure summary, the random interpreter computes a sample S at each program point, as shown in Figure 4.4. A sample is either $\perp$ or a sequence of t states. A state at a program point π is a mapping of program variables (visible at point π) to random symbolic linear expressions in terms of the input variables of the enclosing procedure. We use the notation S_i to denote the i^{th} state in sample S. The random interpreter computes a sample S at each program point from the samples at the immediately preceding program points, and using the summaries computed so far for the called procedures. The transfer functions for the flowchart nodes are described below. After the random interpreter is done interpreting a procedure, it computes the summary of that procedure by simply projecting the sample (or the t states) at the end of the procedure to the output variables of the procedure.

Initialization: The random interpreter starts by initializing the summaries of all procedures, and the samples at all program points except at procedure entry points to $\perp$. The samples at procedure entry points are initialized by setting all input variables x to $\texttt{SEval}'(x)$ in all states.

$$S_i(x) = \texttt{SEval}'(x)$$

Note that $\texttt{SEval}'(x)$ is simply x for the abstractions of linear arithmetic and uninterpreted functions.

Assignment Node: See Figure 4.4 (a).
If the sample S' before the assignment node is $\perp$, then the sample S after the assignment node is defined to be $\perp$. Otherwise, the random interpreter computes S by updating the value of variable x in each state of sample S' as follows.

$$S_i = S'_i[x \leftarrow \texttt{Eval}(e, S'_i)]$$

Non-deterministic Assignment Node: See Figure 4.4 (b).
If the sample S' before the non-deterministic assignment node is $\perp$, then the sample S after the non-deterministic assignment node is defined to be $\perp$. Else, the random interpreter processes the assignment $x :=?$ by transforming each state in the sample S' by setting x to some fresh random value.

$$S_i = S'_i[x \leftarrow v], \text{ where } v = \texttt{SEval}'(y)[\texttt{Rand}()/y]$$

The fresh random value v is obtained from the polynomial $\texttt{SEval}'(y)$ by substituting variable y by a randomly chosen element from $\mathbb{F}_p$.

Non-deterministic Conditional Node: See Figure 4.4 (c).
The random interpreter simply copies the sample S before the conditional node on the two branches of the conditional.

$$S^1 = S \text{ and } S^2 = S$$

Join Node: See Figure 4.4 (d).
If any one of the samples S^1 or S^2 before the join node is $\perp$, the random interpreter assigns the other sample before the join node to the sample S after the join node. Otherwise, the random interpreter selects t random weights $w_1, \ldots, w_t$ and computes the affine join of S^1 and S^2 with respect to those weights to obtain the sample S after the join node.

$$S = \phi_{[w_1,\ldots,w_t]}(S^1, S^2)$$

Procedure Call: See Figure 4.4 (e).
If the sample S' before the procedure call is $\perp$, or if the summary $Y_{P'}$ is $\perp$, then the sample S after the procedure call is defined to be $\perp$. Otherwise the random interpreter executes the procedure call as follows. The random interpreter first generates t fresh random runs $Y_1, \ldots, Y_t$ for procedure P' using the current summary (t runs) for procedure P'. Each fresh run Y_i for procedure P' is generated by taking a random affine combination of the t runs in the summary of procedure P'. This involves choosing random weights $w_{i,1}, \ldots, w_{i,t}$ with the constraint that $w_{i,1} + \cdots + w_{i,t} = 1$, and then doing the following computation. Then,

$$Y_i(x) = \sum_{j=1}^{t} w_{i,j} \times Y_{P',j}(x)$$

The effect of a call to procedure P' is to update the values of the variables that are written to by procedure P'. The random interpreter models this effect by updating the values of these variables using the fresh random runs Y_i (computed above) as follows. Let the input (global) variables of procedure P' be $y_1, \ldots, y_k$. Let $O_{P'}$ denote the set of output (global) variables of procedure P'.

$$S_i(x) = \begin{cases} Y_i(x)[S'_i(y_1)/y_1, \ldots, S'_i(y_k)/y_k] & \text{if } x \in O_{P'} \\ S'_i(x) & \text{otherwise} \end{cases}$$

Phase 2

For the second phase, the random interpreter also maintains a sample S (which is a sequence of t states) at each program point, as in phase 1. However, unlike phase 1, the states in phase 2 map variables to values that do not involve input variables. The samples are computed for each program point from the samples at the preceding program points in the same manner as in phase 1 except for the initialization, which is done as follows:

Initialization: The random interpreter initializes the samples at all program points except at procedure entry points to $\perp$. The sample at the entry point of the `Main` procedure is initialized by setting all input variables x to fresh random values in all states.

$$S_i(x) = \texttt{SEval}'(x)[\texttt{Rand}()/x]$$

As before a fresh random value is obtained from the polynomial $\texttt{SEval}'(x)$ by substituting variable x by a randomly chosen element from $\mathbb{F}_p$.

The sample S at the entry point of any other procedure P is obtained as a random affine combination of all the non-$\perp$ samples at the call sites to P. Let these samples be $S^1, \ldots, S^k$. Then for any input variable x,

$$S_i(x) = \sum_{j=1}^{k} w_{i,j} \times S_i^j(x)$$

where $w_{i,1}, \ldots, w_{i,k}$ are random weights with the constraint that $w_{i,1} + \cdots + w_{i,k} = 1$, for all $1 \leq i \leq t$. This affine combination encodes all the relationships (among input variables of procedure P) that hold in all calls to procedure P.

Verifying and Discovering Equivalences

Let S be the sample computed by the random interpreter at some program point π after fixed-point computation. If $S = \perp$, then the random interpreter declares π to be unreachable. Else, it declares two expressions e_1 and e_2 to be equal at program point π iff for all states S_i in the sample S, $\texttt{Eval}(e_1, S_i) = \texttt{Eval}(e_2, S_i)$.

The process of discovering equivalences at a program point is abstraction specific. For example, Section 2.2.1 describes how to discover linear equality relationships for the abstraction of linear arithmetic. Section 3.2.1 describes how to discover Herbrand equivalences among program sub-expressions for the abstraction of uninterpreted functions.

Optimization

Maintaining a sample explicitly at each program point is expensive (in terms of time and space complexity) and redundant. The optimization of maintaining one global sample for the SSA version of the program, as discussed in Section 2.2.1, applies here as well. Under such an optimization, interpreting an assignment node or a procedure call simply involves updating the values of the modified variables in the global sample. Interpreting a join node involves updating the values of phi-variables at that join point in the global sample.

4.3.2 Error Probability Analysis

In this section, we estimate the error probability of the random interpreter. We show that the random interpreter is complete, i.e. it validates all correct equivalences (property D4). On the other hand, we show that with high probability (over the random choices made by the random interpreter), the random interpreter does not validate a given incorrect equivalence (Theorem 14). We also show that if the `SEval` function does not involve any random variables (e.g., `SEval` function for linear arithmetic), then with high probability (over the random choices made by the random interpreter), the random interpreter validates only correct equivalences (Theorem 14). For the purpose of establishing these results, we first state and prove some useful properties of the samples computed by the random interpreter in phase 1 and phase 2.

Analysis of Phase 1

We first introduce some terminology. We use the term *input context*, or simply *context*, for a procedure P to denote a mapping of input variables of procedure P to polynomials that are linear in program variables. For any context C, let $Abs(C)$ denote the set of equivalences (involving variables that have mappings in C) in the abstract domain that are implied by C, i.e., $Abs(C) = \{e_1 = e_2 \mid \texttt{Eval}(e_1, C) = \texttt{Eval}(e_2, C)\}$. For any polynomial Q, and any context (or state) C, we use the notation $Q[C]$ to denote the polynomial obtained from Q by substituting all variables that have mappings in C by those mappings.

Let π be some program point in procedure P. Let T be some set of paths that lead to π from the entry point of procedure P. We say that an equivalence $e_1 = e_2$ holds at π along paths T in context C iff the weakest precondition of the equivalence $e_1 = e_2$ along all paths in T belongs to the set $Abs(C)$. We denote this by $Holds(e_1 = e_2, T, C)$.

We say that a state ρ entails an equivalence $e_1 = e_2$ in context C, denoted by $\rho \models_C e_1 = e_2$, when $\texttt{Eval}(e_1, \rho)[C] = \texttt{Eval}(e_2, \rho)[C]$. We say that a sample S entails an equivalence $e_1 = e_2$ in context C, denoted by $S \models_C e_1 = e_2$, when all states in S do so.

Let S be the sample computed by the random interpreter (in phase 1) at a program point π in procedure P after analyzing a set of paths T. The following properties hold.

D1. Soundness (Phase 1): Suppose that the `SEval` function does not involve any random variables. With high probability, in all input contexts, S entails only the equivalences that hold at π along the paths analyzed by the random interpreter (i.e., with high probability, for all input contexts C and all equivalences $e_1 = e_2$, $\neg(\mathit{Holds}(e_1 = e_2, T, C)) \Rightarrow \neg(S \models_C e_1 = e_2)$). The error probability $\gamma_1(S)$ (assuming that the samples computed before computation of S satisfy property D1) is bounded above as follows:

$$\gamma_1(S) \leq p^{k_\mathtt{i}} \left(\frac{\alpha^{t-k_\mathtt{v}}}{1-\alpha} \right), \text{ where } \alpha = \frac{3 d_S t}{p(t-k_\mathtt{v})}$$

We use the notation d_S to refer to the number of join points and procedure calls along any path analyzed by the random interpreter immediately after computation of sample S. A formal definition of d_S is given in Appendix C.1.

D2. Completeness (Phase 1): In all input contexts, S entails all equivalences that hold at π along the paths analyzed by the random interpreter (i.e., for all input contexts C and all equivalences $e_1 = e_2$, $\mathit{Holds}(e_1 = e_2, T, C) \Rightarrow S \models_C e_1 = e_2$).

For the purpose of proving property D1, we hypothetically extend the random interpreter to compute a *fully-symbolic state* at each program point, i.e., a state in which variables are mapped to polynomials in terms of the input variables and random weight variables corresponding to join points and procedure calls (see Appendix C.1 for details). A key part of the proof strategy is to prove that the fully-symbolic state at each point captures *exactly* the set of equivalences at that point in any context along the paths analyzed by the random interpreter. In essence, a fully-symbolic interpreter is sound and complete, even though it might be computationally expensive. The proof of this fact is by induction on the number of flowchart nodes analyzed by the random interpreter (Lemma 16 in Appendix C.1). We now prove property D1 using the following two steps.

We first bound the error probability that a sample S with t states does not entail exactly the same set of equivalences as the corresponding fully-symbolic state $\tilde{\rho}$ in *a given context*. The following lemma specifies a bound on this error probability, which we denote by $\gamma_1'(S)$.

Lemma 13 $\gamma_1'(S) \leq \frac{\alpha^{t-k_{\mathrm{v}}}}{1-\alpha}$, *where* $\alpha = \frac{3d_S t}{p(t-k_{\mathrm{v}})}$.

The proof of Lemma 13 is in Appendix C.3.

Next we observe that it is sufficient to analyze the soundness of a sample in a smaller number of contexts (compared to the total number of all possible contexts), which we refer to as a basic set of contexts. If a sample entails exactly the same set of equivalences as the corresponding fully-symbolic state for all contexts in a basic set, then it has the same property for all contexts. Let N denote the number of contexts in any smallest basic set of contexts. The following theorem specifies a bound on N.

Lemma 14 $N \leq p^{k_{\mathrm{i}}}$.

The proof of Lemma 14 is in Appendix C.4.

The probability that a sample S is not sound in any of the contexts is bounded above by the probability that S is not sound in some given context multiplied by the size of any basic set of contexts. Thus, the error probability $\gamma_1(S)$ mentioned in property D1 is bounded above by $\gamma_1'(S) \times N$.

The proof of property D2 is by induction on the number of flowchart nodes analyzed by the random interpreter, and is similar to the proof of completeness of the fully-symbolic state given in Appendix C.1.

Analysis of Phase 2

A sample S computed by the random interpreter in phase 2 at a program point π in procedure P has the following properties.

D3. Soundness (Phase 2): Suppose that the `SEval` function does not involve any random variables. With high probability, S entails only the equivalences that hold at π along the paths analyzed by the random interpreter. The error probability $\gamma_2(S)$ (assuming that the samples computed before computation of sample S satisfy property D3, and all samples computed in phase 1 satisfy property D1) is bounded above as follows:

$$\gamma_2(S) \leq \frac{\alpha^{t-k_{\mathrm{v}}}}{1-\alpha}, \text{ where } \alpha = \frac{3d_S t}{p(t-k_{\mathrm{v}})}$$

d_S is as described in property D1. A formal definition of d_S is given in Appendix C.1.

D4. Completeness (Phase 2): S entails all equivalences that hold at π along the paths analyzed by the random interpreter.

The proof of property D3 is an instantiation of the proof of Lemma 13, and follows from it by choosing the context C to be the identity mapping. The proof of property D4 is an instantiation of the proof of property D2, and follows from it by choosing the context C to be the identity mapping.

Property D4 implies that the random interpreter discovers all valid equivalences. We now use the properties D1 and D3 to prove the following theorem, which establishes a bound on the total error probability of the random interpreter.

Theorem 14 [Probabilistic Soundness Theorem] *Let $H_1 = 1 + k_{\mathtt{v}}(k_{\mathtt{i}} + 1)$ and $H_2 = 1+k_{\mathtt{v}}$. Let $q = nH_1\beta$ and $d = max\{(n_{\mathtt{cp}}+n_{\mathtt{pp}})H_1\beta, (n_{\mathtt{c}}+n_{\mathtt{p}})H_2\beta\}$. Suppose that $p > (3dt)^2$. If* `SEval` *function does not involve any random variables, then the probability that all random samples computed by the random interpreter satisfy only those equivalences that hold at the corresponding program points is at least $1-\frac{2q}{1-\alpha}\alpha^{t-t_0}$, where $\alpha = \frac{3dt}{p(t-k_{\mathtt{v}})}$ and $t_0 = k_{\mathtt{v}}+2k_{\mathtt{i}}$. In general, the probability that the random interpreter does not verify a given false equivalence $e_0 = e_0'$ is bounded below by $1 - \frac{2q}{1-\alpha}\alpha^{t-t_0} - \frac{\delta}{p}$. Here δ refers to the maximum degree of* `SEval`*(e) for any expression e that uses a maximum of $2sH_2(nH_1)^{nH_1}(nH_2)^{nH_2} + s'$ function symbols, where s is the maximum number of function symbols in any assignment node, and s' is the maximum number of function symbols in expressions e_0 and e_0'.*

Proof. It follows from the discussion after Theorem 15 and Theorem 16 in the next section that the random interpreter goes around each loop at most $H_1\beta$ times in phase 1 and $H_2\beta$ times in phase 2 for fixed-point computation. Hence, the random interpreter computes at most $nH_1\beta$ samples in phase 1 and $nH_2\beta$ samples in phase 2. Also, note that the value of d_S in the bounds on the probabilities $\gamma_1(S)$ and $\gamma_2(S)$ (which bound the unsoundness of a sample S) is at most $(n_{\mathtt{cp}} + n_{\mathtt{pp}})H_1\beta$ in phase 1 and $(n_{\mathtt{c}} + n_{\mathtt{p}})H_2\beta$ in phase 2. This implies that the total error probability of the random interpreter for the case when `SEval` function does not involve any random variables is bounded above by $\frac{2q}{1-\alpha}\alpha^{t-t_0}$.

We now prove an upper bound on the probability that the random interpreter with a general `SEval` function validates an incorrect equivalence. Let P_0 be the original program, and let $e_0 = e_0'$ be some equivalence that does not hold in program P_0 at some program point π. Let P_1 be the program obtained from P_0 by replacing all expressions e by `SEval`(e). Let $e_1 =$ `SEval`(e_0) and $e_1' =$ `SEval`(e_0'). It follows from the properties B1, B2 and B3 of the `SEval` function that the equivalence $e_1 = e_1'$ does not hold in program P_1 (at program point π).

Let P_2 be the program obtained from P_1 by substituting all random variables y_j in the `SEval` function by the random values r_j chosen by the random interpreter. Similarly, let $e_2 = e_1[r_j/y_j]$ and $e'_2 = e'_1[r_j/y_j]$. We now show that the probability (over the choice of the random values r_j) that P_2 does not satisfy $e_2 = e'_2$ (at program point π) is bounded above by $\frac{\delta}{p}$. Let P'_1 and P'_2 be the programs without any procedure calls obtained from programs P_1 and P_2 respectively using the procedure inlining technique described in Appendix C.2. The programs P'_1 and P'_2 satisfy the same set of equivalences as the programs P_1 and P_2 respectively at the corresponding points. Each procedure in the programs P'_1 and P'_2 has at most $n_{max} = 2(nH_1)^{nH_1}(nH_2)^{nH_2}$ nodes. Since P'_1 does not satisfy the equivalence $e_1 = e'_1$ (at program point π) it follows from Theorem 16 that there must be a path of length $n_{max}H_2$ (from the entry point of the procedure enclosing point π to π) along which the equivalence $e_1 = e'_1$ is not satisfied. Let ρ_1 be the state obtained by executing the program P'_1 along that path. Note that $e_1[\rho_1] \neq e'_1[\rho_1]$. Let $\rho_2 = \rho_1[r_j/y_j]$. The degrees of the polynomials $e_2[\rho_2]$ and $e'_2[\rho_2]$ are bounded above by δ. It follows from the Schwartz and Zippel's polynomial identity testing theorem that the probability that $e_2[\rho_2] = e'_2[\rho_2]$ is at most $\frac{\delta}{p}$. Hence, the probability that P'_2 (or equivalently P_2) satisfies the equivalence $e_2 = e'_2$ (at program point π) is at most $\frac{\delta}{p}$.

Now suppose that the program P_2 does not satisfy the equivalence $e_2 = e'_2$ (at program point π). Observe that performing random interpretation over program P_0 (using the `SEval` function for the underlying abstraction) to decide the validity of the equivalence $e_0 = e'_0$ (at program point π) is equivalent to performing random interpretation over program P_2 (using the identity `SEval` function) to decide the validity of the equivalence $e_2 = e'_2$ (at program point π). It follows from the result established for the case when `SEval` function does not involve any random variables that the probability that the random interpreter validates the incorrect equivalence $e_2 = e'_2$ in program P_2 (at program point π) is at most $\frac{2q}{1-\alpha}\alpha^{t-t_0}$. The desired result now follows from the union bound on this probability and the probability that program P_2 satisfies the equivalence $e_2 = e'_2$ (at program point π). □

Note that for the case when `SEval` function does not involve any random variables, Theorem 14 gives a bound on the error probability for the process of discovering (all) equivalences; while for the general case, it specifies a bound on the error probability for verification of a (single) given equivalence [2]. The theorem implies that for probabilistic

[2]The error probability for verification of m equivalences can be obtained by multiplying the error probability for verification of one equivalence by m.

soundness we need to choose t to be greater than $k_{\mathtt{v}} + 2k_{\mathtt{i}}$. It may be possible to prove better bounds on t for specific abstractions (e.g., we only require $t > 6$ for the case of unary uninterpreted functions, as discussed in Section 4.4.2). For the case when `SEval` function does not involve any random variables, p needs to be greater than $(3dt)^2$. However, for the general case, we need to choose p to be greater than δ, which implies that we need to perform arithmetic with numbers that require $O(\log \delta)$ bits for representation. For example, the value of δ for the theory of unary uninterpreted functions is $2sH_2(nH_1)^{nH_1}(nH_2)^{nH_2} + s'$, and this implies that the arithmetic should be performed with numbers that require $O(nk_{\mathtt{v}} \log n)$ bits for representation. However, we feel that this analysis is very conservative, and experiments suggest that 32-bit primes are good enough in practice.

4.3.3 Fixed-point Computation

The notion of loop that we consider for fixed-point computation is that of a maximal strongly connected component (SCC). For defining SCCs in a program in an inter-procedural setting, we consider the directed graph representation of a program that has been referred to as supergraph in the literature [RHS95]. This directed graph representation consists of a collection of flowcharts, one for each procedure in the program, with the addition of some new edges. For every edge to a call node, say from node η_1 to call node η_2 with the call being to procedure P, we add two new edges: one from node η_1 to entry node of procedure P, and the other from exit node of procedure P to node η_2. Now consider the DAG of SCCs of this directed graph representation of the program. Note that an SCC in this DAG may contain nodes of more than one procedure [3] (in which case it contains all nodes of those procedures).

In both phase 1 and phase 2, the random interpreter processes all SCCs in the DAG in a top-down manner. It goes around each SCC until a fixed point is reached. In phase 1, a sample computed by the random interpreter represents sets of equivalences, one for each context. A fixed point is reached for an SCC in phase 1, if for all points π in the SCC and for all contexts C (for the procedure enclosing point π), the set of equivalences at π in context C has stabilized. In phase 2, a sample computed by the random interpreter represents a set of equivalences; and a fixed point is reached for an SCC, if for all points π in the SCC, the set of equivalences at π has stabilized. Let H_1 and H_2 be the upper bounds

[3]This happens when the call graph of the program contains a maximal strongly connected component of more than one node.

on the number of iterations required to reach a fixed point across any SCC in phase 1 and 2 respectively.

Theorem 15 [Fixed Point Theorem for Phase 1] *Let $\tilde{\rho}^1, \ldots, \tilde{\rho}^m$ be the fully-symbolic states computed by the random interpreter in phase* 1 *at some point π inside a loop in successive iterations of that loop such that $\tilde{\rho}^i$ does not imply the same set of equivalences as $\tilde{\rho}^{i+1}$ in some context. Then, $m \leq H_1$, where $H_1 = 1 + k_{\mathtt{v}}(k_{\mathtt{i}} + 1)$.*

Proof. We first introduce some notation. Let ρ be any state that maps variables $x_1, \ldots, x_a$ to polynomials that are linear in input variables $y_1, \ldots, y_b$. Let $\rho(x_j) = \left(\sum_{i=1}^{b} v_{i+(j-1)(b+1)} y_i\right) + v_{j(b+1)}$, where the polynomials v_i do not involve the input variables $y_1, \ldots, y_b$. We use the notation $R(\rho)$ to denote the vector $(v_1, \ldots, v_{a(b+1)}, 1)$. For any fully-symbolic state $\tilde{\rho}$, we use the notation $\mathcal{V}(\tilde{\rho})$ to denote the smallest vector space generated by $\{R(\tilde{\rho})[v_i/w_i] \parallel v_i \in \mathbb{F}_p\}$ over the field $\mathbb{F}_p$, where $R(\tilde{\rho})[v_i/x_i]$ denotes the vector obtained from $R(\tilde{\rho})$ by replacing all weight variables w_i by some choices of elements v_i from $\mathbb{F}_p$.

Consider the vector spaces $\mathcal{V}(\tilde{\rho}^i)$ and $\mathcal{V}(\tilde{\rho}^{i+1})$. Note that $\mathcal{V}(\tilde{\rho}^i)$ is included in $\mathcal{V}(\tilde{\rho}^{i+1})$ since $\tilde{\rho}^i$ is an instantiation of $\tilde{\rho}^{i+1}$. Also, note that $\mathcal{V}(\tilde{\rho}^i) \neq \mathcal{V}(\tilde{\rho}^{i+1})$ since $\tilde{\rho}^i$ does not imply the same set of equivalences as $\tilde{\rho}^{i+1}$ in some context. Hence, $\mathit{Rank}(\mathcal{V}(\tilde{\rho}^i)) < \mathit{Rank}(\mathcal{V}(\tilde{\rho}^{i+1}))$. Note that $\mathit{Rank}(\mathcal{V}(\tilde{\rho}^1)) \geq 1$ and $\mathit{Rank}(\mathcal{V}(\tilde{\rho}^m)) \leq 1 + k_{\mathtt{v}}(k_{\mathtt{i}} + 1)$. Hence, $m \leq 1 + k_{\mathtt{v}}(k_{\mathtt{i}} + 1)$. □

Theorem 16 [Fixed Point Theorem for Phase 2] *Let $\tilde{\rho}^1, \ldots, \tilde{\rho}^m$ be the fully-symbolic states computed by the random interpreter in phase* 2 *at some point π inside a loop in successive iterations of that loop such that $\tilde{\rho}^i$ does not imply the same set of equivalences as $\tilde{\rho}^{i+1}$ in some context. Then, $m \leq H_2$, where $H_2 = 1 + k_{\mathtt{v}}$.*

The proof of Theorem 16 is similar to the proof of Theorem 15 and follows from the observation that $\mathit{Rank}(\mathcal{V}(\tilde{\rho}^m)) \leq 1 + k_{\mathtt{v}}$.

We have described a worst-case bound on the number of iterations required to reach a fixed point. However, we do not know if there is an efficient way to detect a fixed point since the random interpreter works with randomized data-structures. The random interpreter can use the strategy of iterating around a loop (SCC) $H_1\beta$ times in phase 1 and for $H_2\beta$ times in phase 2, where β is the number of back-edges in the loop. Note that this guarantees that a fixed point will be reached. This is because if a fixed point is not reached, then the set of equivalences corresponding to at least one of the samples (computed by the random interpreter) at the target of the back-edges must change, and it follows from Theorem 15 and Theorem 16 that there can be at most H_1 or H_2 such changes at each program point in phase 1 or phase 2 respectively.

4.3.4 Computational Complexity

We assume that the random interpreter performs the optimization of maintaining one global state in the SSA version of the program as discussed in Section 4.3.1. Under that optimization, a join operation reduces to processing phi-assignments at that join point. We also assume unit cost for each arithmetic operation and that the size of polynomial $\texttt{SEval}(e)$ is linear in size of expression e. We estimate the running time of the random interpreter for phase 1, which dominates the running time for phase 2. The cost of processing each assignment, both phi and non-phi, is $O(k_{\texttt{i}}t)$. The cost of processing a procedure call is $O(k_{\texttt{i}}k_{\texttt{o}}t^2)$. For fixed-point computation, the random interpreter goes around each loop at most $H_1\beta$ times. Assuming β to be a constant, the running time of the random interpreter is $O(n_{\texttt{s}}H_1k_{\texttt{i}}t + n_{\texttt{p}}H_1k_{\texttt{i}}k_{\texttt{o}}t^2)$. It follows from Theorem 14 that for probabilistic soundness, we need to choose t to be greater than $k_{\texttt{v}} + 2k_{\texttt{i}}$. This yields a total complexity of $O(n_{\texttt{s}}k_{\texttt{v}}^2k_{\texttt{i}}^2 + n_{\texttt{p}}k_{\texttt{v}}^3k_{\texttt{i}}^2k_{\texttt{o}})$ for the random interpreter.

If we regard $k_{\texttt{i}}$ and $k_{\texttt{o}}$ to be constants, since they denote the size of the interface between procedure boundaries and are supposedly small, and the number of procedure call nodes $n_{\texttt{p}}$ to be significantly smaller than the number of assignment nodes $n_{\texttt{s}}$, then the complexity of the random interpreter reduces to $O(n_{\texttt{s}}k_{\texttt{v}}^2)$, which is linear in the size of the program and quadratic in the maximum number of visible program variables at any program point.

4.4 Special Cases

4.4.1 Linear Arithmetic

In this section we discuss the use of inter-procedural random interpretation to discover linear equalities among program variables that take rational values. The basic strategy is to discover the linear equalities among program variables over a randomly chosen prime field $\mathbb{F}_p$ (rather than the infinite field of rationals) using the `SEval` function described

```
P_0(x)  = { return 2x; }
P_i(x)  = { y := P_{i-1}(x); return P_{i-1}(y); }
P'_0(x) = { return 2x; }
P'_i(x) = { y := P'_{i-1}(x); return P'_{i-1}(y); }
Main()  = { y_1 := P_m(0); y_2 := P'_m(0); assert(y_1 = y_2); }
```

Figure 4.5: A program of size $O(m)$ for which any deterministic summary based inter-procedural analysis requires $\Omega(2^m)$ space and time for manipulating arithmetic constants (assuming standard binary representation). The program contains $2m+3$ procedures `Main`, P_i and P'_i for $i \in \{0, \ldots, m\}$. A randomized analysis does not have this problem.

in Section 4.1.1. This is followed by mapping back the discovered equalities to the field of rationals using the technique described in Section 2.2.1.

The motivation for first solving the problem over a randomly chosen prime field (as opposed to directly solving over the infinite field of rationals) comes from the need to avoid manipulating large numbers. For example, consider the program shown in Figure 4.5. Any summary-based approach for discovering linear equalities will essentially compute the following summaries for procedures P_m and P'_m:

$$P_m(x) = 2^{2^m}x$$
$$P'_m(x) = 2^{2^m}x$$

Note that representing the arithmetic constant 2^{2^m} using standard decimal notation requires $\Omega(2^m)$ digits. The problem of manipulating large numbers can be avoided by performing computations over a small finite field $\mathbb{F}_p$, where prime p is chosen randomly. However, this results in some additional error probability, which we discuss below.

Error Probability Analysis

The error probability of the random interpreter can be decomposed into two parts. The special case in Theorem 14 gives the error probability of the random interpreter assuming that the linear equalities are to be discovered over the prime field $\mathbb{F}_p$. We now estimate the remaining error probability, which results from reducing the problem of finding linear equalities over rationals to finding them over $\mathbb{F}_p$.

First observe that the given program can be converted into an equivalent program (in the sense that both programs satisfy the same set of linear equalities at corresponding points) that does not use any procedure calls, and it can be obtained using the procedure inlining technique described in Appendix C.2. Each procedure in the equivalent new program, has at most $n_{max} = 2(nH_1)^{nH_1}(nH_2)^{nH_2}$ nodes. Now, we can use the results developed in Section 2.2.2 and Section 2.3 over the equivalent new program to estimate the additional error probability. Theorem 3 and Theorem 4 stated in Section 2.2.2 state this error probability, which is a function of the size of the set from which the prime p should be chosen randomly. The parameter b_m used in the statement of Theorem 3 and Theorem 4 can be estimated using the analysis in Section 2.3. Since each procedure in the equivalent new program has at most n_{max} nodes (and hence at most n_{max} assignment nodes, and n_{max} variables), we have $b_m \leq 1 + n_{max}(n_{max} + 1)(\log s + \log c_m)$ (where s and c_m are as defined in Section 2.2.2). This implies that for probabilistic soundness the prime number p for performing arithmetic should be chosen randomly from the set $[1, p_m]$, where p_m requires $O(\log n_{max}) = O(nk_{\mathtt{v}}k_{\mathtt{i}} \log n)$ bits for representation. However, we feel that this analysis is conservative. Experiments discussed in Section 4.6 suggest that even 32-bit primes do not yield any error in practice.

Computational Complexity

As discussed in Section 4.3.4, the inter-procedural random interpreter has a complexity of $O(n_{\mathtt{s}}k_{\mathtt{v}}k_{\mathtt{i}}^2 t + n_{\mathtt{p}}k_{\mathtt{v}}k_{\mathtt{i}}^2 k_{\mathtt{o}} t^2)$ (assuming unit cost for each arithmetic operation). It follows from Theorem 14 that for probabilistic soundness, we need to choose t to be greater than $k_{\mathtt{v}} + 2k_{\mathtt{i}}$. However, we feel that our analysis for probabilistic soundness is conservative. Experiments discussed in Section 4.6 suggest that even $t = 3$ does not yield any error in practice if we want to verify equalities (among any number of program variables), or discover equalities between 2 program variables.

4.4.2 Uninterpreted Functions

In this section, we discuss the use of inter-procedural random interpretation for discovering Herbrand equivalences among program sub-expressions that have been abstracted using unary uninterpreted functions. This abstraction is useful for modeling fields of data-structures and can be used to compute must-alias information.

Note that we restrict our attention to unary uninterpreted functions (instead of considering the more general binary uninterpreted functions). This is because in the case of binary uninterpreted functions, expressions are mapped to vectors rather than scalars. The size of these vectors is linearly proportional to the depth of any expression computed by the program along any acyclic path as discussed in Chapter 3. In an inter-procedural setting, the depth of such expressions can be exponential in the size of the program. Hence, unless we can prove the conjecture that the size of the vectors need only be logarithmic in the size of the program (as mentioned in Section 3.2.4), the complexity of processing each node will be exponential in the size of the program, which is perhaps not any worst-case better than a non-summary based inter-procedural analysis (which involves reducing the problem of inter-procedural analysis to intra-procedural analysis by doing procedure inlining as described in Appendix C.2). Since we are interested in polynomial-time complexity algorithms in this dissertation, we leave out the discussion of the complexity of the inter-procedural analysis for binary uninterpreted functions. However, the inter-procedural random interpretation technique described in this chapter is applicable to reasoning about binary uninterpreted functions too.

Error Probability Analysis

Since the `SEval` function for unary uninterpreted functions contains random variables, the general case in Theorem 14 applies, which specifies a bound on the error probability for verification of one equivalence. The total error probability of the random interpreter is given by the product of this error probability (for verification of one equivalence) with the number of equivalences between program sub-expressions verified by the random interpreter.

For probabilistic soundness, Theorem 14 requires choosing t to be greater than $k_{\mathtt{v}} + 2k_{\mathtt{i}}$. However, for the specific case of unary uninterpreted functions, we require t to be only greater than 6. This is because of the following reason. Observe that any equivalence in the abstraction of unary uninterpreted functions involves only 2 program variables. Also, observe that the validity of any equivalence (at any program point) depends on the relationship between at most 2 input variables of the enclosing procedure. Hence, the proof of Theorem 14 can be specialized to the specific case of unary uninterpreted functions by substituting $k_{\mathtt{v}} = 2$ and $k_{\mathtt{i}} = 2$, which yields the desired constraint that t need only be greater than 6.

Computational Complexity

It follows from the above discussion that for probabilistic soundness, we need to choose t to be greater than 6. This yields a total complexity of $O(n_{\mathtt{s}} k_{\mathtt{v}} k_{\mathtt{i}}^2 + n_{\mathtt{p}} k_{\mathtt{v}} k_{\mathtt{i}}^2 k_{\mathtt{o}})$ for the random interpreter (assuming unit cost for each arithmetic operation).

4.5 Related Work

Precise inter-procedural analysis is provably harder than intra-procedural analysis [Rep96]. There is no general recipe for constructing a precise and efficient inter-procedural analysis from just the corresponding intra-procedural analysis. The functional approach proposed by Sharir and Pnueli [SP81] is limited to finite lattices of dataflow facts. Sagiv, Reps and Horwitz have generalized the Sharir-Pnueli framework to build context-sensitive analyses, using graph reachability [RHS95], even for some kind of infinite domains. They successfully applied their technique to detect linear constants inter-procedurally [SRH96]. However, their generalized framework requires appropriate distributive transfer functions as input. There seems to be no obvious way to automatically construct context-sensitive transfer functions from just the corresponding intra-procedural analysis. In this chapter, we have described a general procedure for lifting intra-procedural random interpretation based analyses to perform a precise and efficient inter-procedural analysis.

Linear Arithmetic

Recently, Muller-Olm and Seidl gave a deterministic algorithm (MOS) that discovers all linear equalities in programs that have been abstracted using non-deterministic conditionals [MOS04b]. The MOS algorithm is also based on computing summaries of procedures. However, their summaries are deterministic and consist of linearly independent runs of the program. The program shown in Figure 4.6 illustrates the difference between the deterministic summaries computed by MOS algorithm and the randomized summaries computed by our algorithm. The MOS algorithm maintains the (linearly independent) real runs of the program, and it may have to maintain as many as $k_{\mathtt{v}}(k_{\mathtt{i}} + 1)$ runs. The runs maintained by our algorithm are fictitious as they do not arise in any concrete execution of the program; however they have the property that (with high probability over the ran-

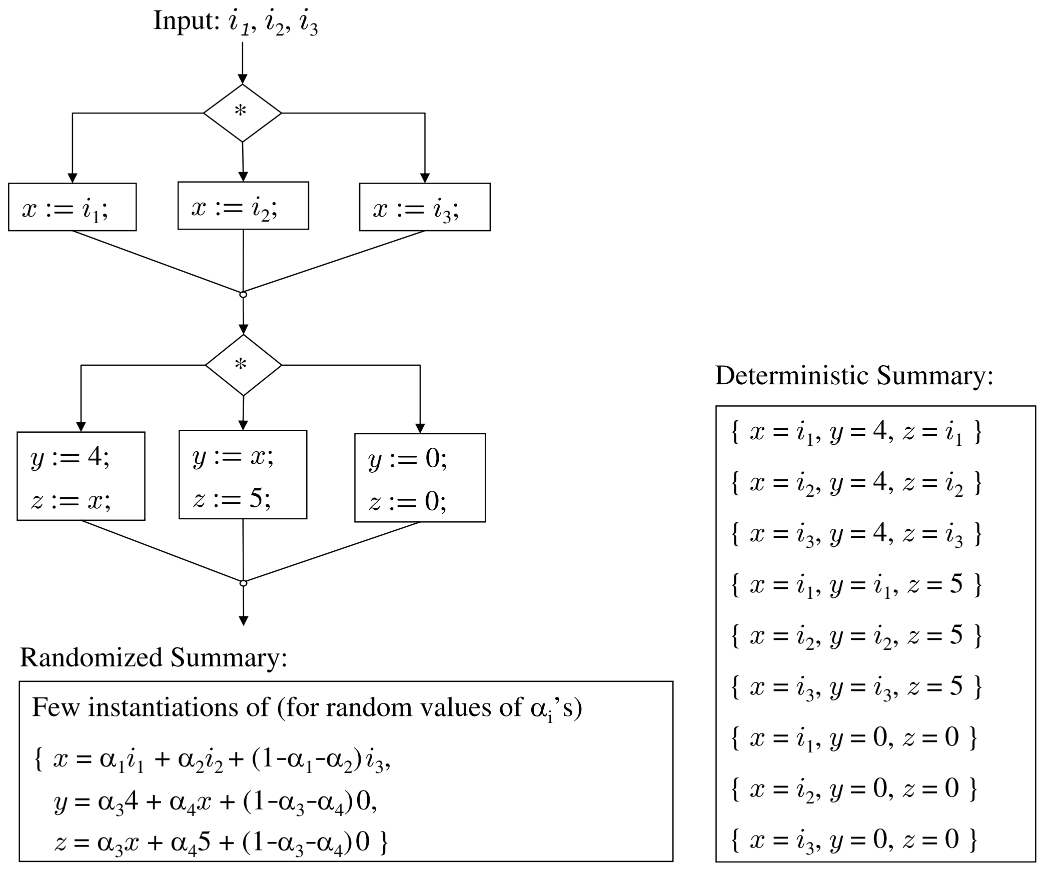

Input: i_1, i_2, i_3
*
$x := i_1;$
$x := i_2;$
$x := i_3;$
*
$y := 4;$ $z := x;$
$y := x;$ $z := 5;$
$y := 0;$ $z := 0;$
Randomized Summary:
Few instantiations of (for random values of α_i's)
$\{ x = \alpha_1 i_1 + \alpha_2 i_2 + (1-\alpha_1-\alpha_2) i_3,$
$y = \alpha_3 4 + \alpha_4 x + (1-\alpha_3-\alpha_4) 0,$
$z = \alpha_3 x + \alpha_4 5 + (1-\alpha_3-\alpha_4) 0 \}$
Deterministic Summary:
$\{ x = i_1, y = 4, z = i_1 \}$
$\{ x = i_2, y = 4, z = i_2 \}$
$\{ x = i_3, y = 4, z = i_3 \}$
$\{ x = i_1, y = i_1, z = 5 \}$
$\{ x = i_2, y = i_2, z = 5 \}$
$\{ x = i_3, y = i_3, z = 5 \}$
$\{ x = i_1, y = 0, z = 0 \}$
$\{ x = i_2, y = 0, z = 0 \}$
$\{ x = i_3, y = 0, z = 0 \}$

Figure 4.6: Illustration of the difference between the deterministic summary computed by MOS algorithm and the randomized summary computed by our algorithm.

dom choices made by the algorithm) they entail exactly the same set of equivalences in all contexts as do the real runs. Our algorithm needs to maintain only a few runs. The conservative theoretical bounds show that more than $k_{\mathtt{v}} + 2k_{\mathtt{i}}$ runs are required, while experiments suggest that even 3 runs are good enough (if we want to verify linear equalities, or discover linear equalities between 2 program variables).

The authors have proved a complexity of $O(nk_{\mathtt{v}}^8)$ for the MOS algorithm assuming a unit cost measure for arithmetic operations. It turns out that the arithmetic constants that arise in MOS algorithm may be large enough that $\Omega(2^n)$ bits for required for representing constants, and hence $\Omega(2^n)$ time is required for performing a single arithmetic operation. The program shown in Figure 4.5 (on page 98) illustrates such an exponential behavior of MOS algorithm. The MOS algorithm can also use the technique of avoiding large arithmetic constants by performing arithmetic modulo a randomly chosen prime, as described in Section 4.4.1. However this makes MOS a randomized algorithm; and the complexity of our randomized algorithm remains better than that of MOS. It is not clear if there exists a polynomial time deterministic algorithm for this problem.

Sagiv, Reps and Horwitz gave an efficient algorithm (SRH) to discover linear constants inter-procedurally in a program [SRH96]. Their analysis considers only those affine assignments whose right hand sides contain at most one occurrence of a variable. However, our analysis is more precise as it treats all affine assignments in a precise manner, and also it discovers all linear equalities (not just constants). In Section 4.6, we experimentally compare the precision and the running time of our analysis with that of SRH algorithm.

The first intra-procedural analysis for discovering linear equalities was given by Karr way back in 1976 [Kar76]. The fact that it took several years to obtain an inter-procedural analysis for discovering all linear relationships in programs that have been abstracted using linear arithmetic assignments demonstrates the complexity of inter-procedural analysis.

Uninterpreted Functions

Recently, Müller-Olm, Seidl, and Steffen have given an algorithm to detect Herbrand equalities in an inter-procedural setting [MOSS05]. Their algorithm is complete (i.e., it detects all valid Herbrand equalities) for side-effect-free procedures that have only one return value. Their algorithm can also detect all Herbrand constants. In contrast, our random interpretation based inter-procedural analysis detects all equivalences without any restriction on the number of return values, or global values affected by a procedure. However, our algorithm has a polynomial time complexity bound only for unary uninterpreted functions.

4.6 Experiments

In this dissertation, we have expanded the body of theoretical evidence that randomized algorithms have certain advantages, such as simpler implementations and better computational complexity, over deterministic ones. We now describe our experience experimenting with some of these algorithms. The goals of these experiments are threefold: (1) measure experimentally the soundness probability and its variation with certain parame-

t	$p = 983$			$p = 65003$			$p = 268435399$		
	$E_{\text{x=c}}$	$E_{\text{x=y}}$	E_{dep}	$E_{\text{x=c}}$	$E_{\text{x=y}}$	E_{dep}	$E_{\text{x=c}}$	$E_{\text{x=y}}$	E_{dep}
2	1.7	0.2	95.5	0.1	0	95.5	0	0	95.5
3	0	0	64.3	0	0	3.2	0	0	0
4	0	0	0.2	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0

$E_{\text{x=c}}$:	% of incorrect variable constants reported
$E_{\text{x=y}}$:	% of incorrect variable equalities reported
E_{dep} :	% of incorrect dependent induction variables reported

Table 4.1: Percentage of incorrect relationships of different kinds discovered by the inter-procedural random interpreter as a function of the number of runs t and the randomly chosen prime p on a collection of programs, which are listed in Table 4.2. For example, with $t = 2$ and $p = 983$, the random interpreter discovered 3501 variable constants, of which 59 were incorrect; hence, $E_{x=c} = \frac{59}{3501-59} \times 100 \approx 1.7\%$. The total number of correct relationships discovered were 3442 variable constants, 4302 variable equalities, and 50 dependent induction variables.

ters of the algorithm, (2) measure the running time and effectiveness of the inter-procedural version of the algorithm, and compare it to the intra-procedural version, and (3) perform a similar comparison with a deterministic inter-procedural algorithm.

We ran all experiments on a Pentium 1.7GHz machine with 1Gb of memory. We used a number of programs, up to 28,000 lines long, some from the SPEC95 benchmark suite, and others from similar measurements in previous work [SRH96]. We measured running time using enough repetitions to avoid timer resolution errors.

We have implemented the inter-procedural algorithm described in this paper, in the context of the linear equalities domain. The probability of error grows with the complexity of the relationships we try to find, and shrinks with the increase in number of runs and the size of the prime number used for modular arithmetic. The last two parameters have a direct impact on the running time. [4]

We first ran the inter-procedural randomized analysis on our suite of programs, using a variable number of runs, and prime numbers of various sizes. We consider here equalities with constants (x=c), variable equalities (x=y), and linear induction variable dependencies among variables used and modified in a loop (dep). [5] Table 4.1 shows the number

[4]For larger primes, the arithmetic operations cannot be performed directly with machine arithmetic.

[5]We found many more linear dependencies, but report only the induction variable ones because those have a clear use in compiler optimization.

of erroneous relationships reported in each case, as a percentage of the total relationships found for the corresponding kind.

These results are for programs with hundreds of variables; and our analysis for probabilistic soundness requires $t > k_{\mathtt{v}} + 2k_{\mathtt{i}}$, yet in practice we do not notice any errors for $t \geq 4$. Similarly, our theoretical estimates of the error probability when using small primes are also pessimistic. With the largest prime shown in Table 4.1, we did not find any errors if we use at least 3 runs. [6] In fact, for the problem of discovering simpler kinds of equalities (variable constants $x = c$, variable equalities $x = y$), we do not observe any errors for $t = 2$. This is in fact the setup that we used for the experiments described below that compare the precision and cost (in terms of time) of the randomized inter-procedural analysis with that of randomized intra-procedural analysis and deterministic inter-procedural analysis.

The first set of columns in Table 4.2 show the results of the inter-procedural randomized analysis for a few benchmarks with more than 1000 lines of code each. The column headings are explained in the caption. We ran the algorithm with both $t = 2$ and $t = 3$, since the smaller value is faster and sufficient for discovering equalities between variables and constants. As expected, the running time increases linearly with t. The noteworthy point here is the number of relationships found between the input variables of a procedure.

In the second set of columns in Table 4.2 we show how many fewer relationships of each kind are found by the intra-procedural randomized analysis, and how much faster that analysis is, when compared to the inter-procedural one. The intra-procedural analysis obviously misses all of the input relationships and consequently misses some internal relationships as well, but it is much faster. The loss of effectiveness results (when performing an inter-procedural analysis as compared to an intra-procedural analysis) are similar to those reported in [SRH96]. Whether the additional information generated by the inter-procedural analysis is worth the extra implementation and compile-time cost will depend on how that information is to be used. For compiler optimization it is likely that intra-procedural results are good enough, but perhaps for applications such as program verification the extra cost might be worth paying.

Finally, we compare our inter-procedural random interpretation based algorithm with an inter-procedural deterministic algorithm. We have implemented and experimented

[6] With only 2 runs, we find a linear relationship between any pair of variables, as expected.

Program	Size	Random Inter.						Random Intra.				Det. Inter.	
		inp	x=c	x=y	dep	T_2	T_3	Δ x=y	Δ x=c	Δ dep	R_3	Δ inp	R_2
go	29K	63	1700	796	6	47.3	70.4	170	260	3	107	17	1.9
ijpeg	28K	31	825	851	12	3.8	5.7	34	1	9	24	3	2.3
li	23K	53	392	2283	9	34.0	51.4	160	1764	6	756	20	1.3
gzip	8K	49	525	372	2	2.0	3.05	200	12	1	39	6	2.0
bj	2K	0	117	9	0	1.2	1.8	0	0	0	11	0	2.3
linpackc	2K	14	86	16	1	0.07	0.11	17	1	1	9	0	1.8
sim	2K	3	117	296	0	0.35	0.54	3	11	0	22	0	1.7
whets	1K	9	80	2	6	0.03	0.05	17	1	0	9	0	1.5
flops	1K	0	52	4	4	0.02	0.03	0	0	0	22	0	2.0

Random Inter.:	Randomized Inter-procedural Analysis
Random Intra.:	Randomized Intra-procedural Analysis
Det. Inter.:	Deterministic Inter-procedural Analysis

Size:	# of lines of C-code
inp:	# of linear relationships among input variables at procedure entry points
x=c:	# of variables equal to constant values
x=y:	# of variable equalities
dep:	# of dependent loop induction variables
T_i:	Time (in seconds) for $t = i$ runs
Δ k:	Difference of # of relationships of kind k found by Random Inter and given algorithm
R_i:	Ratio of time with time T_i of Random Inter.

Table 4.2: Comparison of precision and efficiency between the randomized inter-procedural, randomized intra-procedural, and deterministic inter-procedural analyses on SPEC benchmarks.

with the SRH algorithm [SRH96], and the results are shown in the third set of columns in Table 4.2. SRH is less precise than our algorithm, in that it searches only for equalities with constants ($x = c$). It does indeed find all such equalities that we also find. In theory, there are equalities with constants that we can find but SRH cannot, because they are consequences of more complex linear relationships. However, the set of benchmarks that we have looked at does not seem to have any such hard-to-find equalities. For comparison with this algorithm, we used $t = 2$, which is sufficient for finding equalities of the form $x = c$ and $x = y$. However, we find a few more equalities between the input variables (Δ inp), and numerous equalities between local variables, which SRH does not attempt to find. On average, SRH is 1.5 to 2.3 times faster than our algorithm. Again, the cost may be justified by the expanded set of relationships that we discover.

A fairer comparison would have been with the MOS algorithm [MOS04b], which is as precise as our inter-procedural randomized algorithm. However, implementing this algorithm seems quite a bit more complicated than either of our algorithm or SRH. We also could not obtain an implementation from anywhere else. Furthermore, we speculate that due to the fact that MOS requires data structures whose size is $O(k_{\mathrm{v}}^4)$ at every program point, it will not fare well on the larger examples that we have tried, which have hundreds of variables and tens of thousands of program points. Another source of bottleneck may be the complexity of manipulating large constants that may arise during its analysis.

Creativity is the ability to introduce order into the randomness of nature.

Eric Hofer.

Chapter 5

Combining Program Analyses

The random interpretation scheme described in Chapter 2 reasons about the abstraction of linear arithmetic while the one described in Chapter 3 reasons about the abstraction of uninterpreted functions. It is natural to ask if we can combine the two random interpretation schemes to reason about the combined abstraction.

We can state the problem more formally as follows. Suppose the flowchart representation of a procedure consists of nodes of the kind shown in Figure 5.1. The random interpretation scheme described in Chapter 2 can discover linear equalities when the expression language in the program consists of the following:

$$e ::= x \quad | \quad c \quad | \quad e_1 \pm e_2 \quad | \quad c \times e \tag{5.1}$$

The random interpretation scheme described in Chapter 3 can discover Herbrand equivalences when the expression language in the program consists of the following (For simplicity, we restrict the expression language here to consist of only unary uninterpreted functions):

$$e ::= x \quad | \quad F(e) \tag{5.2}$$

It is interesting to consider if there is a random interpretation scheme that can discover equivalences between mixed expressions when the expression language in the program consists of the following:

$$e ::= x \quad | \quad c \quad | \quad e_1 \pm e_2 \quad | \quad c \times e \quad | \quad F(e) \tag{5.3}$$

We know from Section 4.1.1 that a random interpreter is parametrized by an `SEval` function. If there exists an `SEval` function for an expression language, then there exists

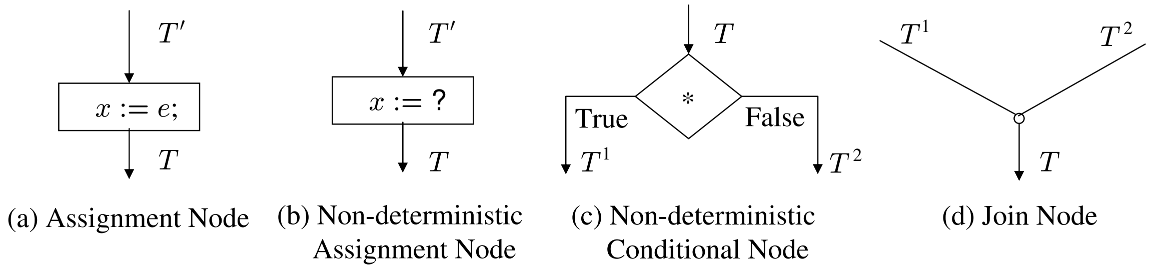


Figure 5.1: Flowchart nodes considered in the combination of program analyses.

a random interpretation scheme to reason about programs with that expression language. Hence, the question that we are asking is whether there exists an SEval function for the expression language described above in Equation 5.3.

It is interesting to consider if we can naively combine the SEval functions for the expressions languages for linear arithmetic and uninterpreted functions to obtain the SEval function for the combined expression language. The SEval function for the expression language in Equation 5.1 is simply:

$$\texttt{SEval}(e) = e$$

The SEval function for the expression language in Equation 5.2 is:

$$\begin{aligned}\texttt{SEval}(x) &= x \\ \texttt{SEval}(F(e)) &= r_1 \times \texttt{SEval}(e) + r_2\end{aligned}$$

A naive combination of the above two SEval functions yields the following SEval function for the combined expression language:

$$\begin{aligned}\texttt{SEval}(x) &= x \\ \texttt{SEval}(c) &= c \\ \texttt{SEval}(e_1 \pm e_2) &= \texttt{SEval}(e_1) \pm \texttt{SEval}(e_2) \\ \texttt{SEval}(c \times e) &= c \times \texttt{SEval}(e) \\ \texttt{SEval}(F(e)) &= r_1 \times \texttt{SEval}(e) + r_2\end{aligned}$$

Unfortunately, such a naive combination of the two SEval functions does not satisfy the soundness property (property B1 in Section 4.1.1, which states that an SEval function does not introduce any new equivalences). For example, consider the two unequal expressions $e_1 = F(a+c) - F(b+c)$ and $e_2 = F(a) - F(b)$. The reader can easily verify that $\texttt{SEval}(e_1) = \texttt{SEval}(e_2)$.

We now present a result that implies that there cannot be any efficient `SEval` function for the combined expression language of linear arithmetic and uninterpreted functions. This result is quite surprising given that there are efficient techniques for combining decision procedures for the theories of linear arithmetic and uninterpreted functions [NO79].

5.1 Hardness of the combination

In this section, we show that the problem of verifying equivalences among program expressions when the expression language of the program is given by Equation 5.3 (and the flowchart representation of the program consists of the nodes shown in Figure 5.1) is NP-hard.

Consider the program shown in Figure 5.2. The assert statement in the program is true iff the input boolean formula ψ is unsatisfiable. Note that ψ is unsatisfiable iff at least one of its clauses remains unsatisfiable in any truth value assignment to its variables, or equivalently, $g \in \{0, \ldots, m-1\}$ in all executions of the procedure `IsUnSatisfiable`, which non-deterministically sets variables in ψ to some truth value. The assert statement in procedure $\texttt{Check}(g, m)$ is true iff $g \in \{0, \ldots, m-1\}$, as stated in Lemma 15. The key idea in the proof of this lemma is that a disjunctive assertion of the form $x = a \vee x = b$ can be encoded as the non-disjunctive assertion $F(a) + F(b) = F(x) + F(a+b-x)$ (which uses expressions in the combination of linear arithmetic and uninterpreted functions).

Lemma 15 *The assert statement in* $\texttt{Check}(g, m)$ *is true iff* $g \in \{0, \ldots, m-1\}$.

Proof. The following properties hold for all $0 \leq i \leq m-1$.

E1. If $0 \leq j \leq i$, then $h_{i,j} = h_{i,0}$.

E2. If $g \in \{0, \ldots, m-1\}$, then $h_i = h_{i,g}$.

E3. If $g \notin \{0, \ldots, m-1\}$, then h_i cannot be expressed as linear combination of $\{h_{i,j} \parallel 0 \leq j \leq m-1\}$.

```
IsUnSatisfiable($\psi$)
    % Let formula $\psi$ has $k$ variables $x_1, \ldots, x_k$
    %               and $m$ clauses numbered 1 to $m$.
    % Let variable $x_i$ occur in positive form in clauses # $A_i[0], \ldots, A_i[c_i]$
    %                    and in negative form in clauses # $B_i[0], \ldots, B_i[d_i]$.
    for $i$ = 1 to $m$ do
        $e_i := 0$; % $e_i$ represents whether clause $i$ is satisfiable or not.
    for $i$ = 1 to $k$ do
        if (*) then % set $x_i$ to true
            for $j$ = 0 to $c_i$ do
                $e_{A_i[j]} := 1$;
        else % set $x_i$ to false
            for $j$ = 0 to $d_i$ do
                $e_{B_i[j]} := 1$;
    $g := e_1 + e_2 + \ldots + e_m$; % Count how many clauses have been satisfied.
    Check($g, m$);


Check($g, m$)
    % This procedure checks whether $g \in \{0, \ldots, m-1\}$.
    $h_0 := F(g)$;
    for $j$ = 0 to $m-1$ do
        $h_{0,j} := F(j)$;
    for $i$ = 1 to $m-1$ do
        $s_{i-1} := h_{i-1,0} + h_{i-1,i}$;
        $h_i := F(h_{i-1}) + F(s_{i-1} - h_{i-1})$;
        for $j$ = 0 to $m-1$ do
            $h_{i,j} := F(h_{i-1,j}) + F(s_{i-1} - h_{i-1,j})$;
    Assert($h_{m-1} = h_{m-1,0}$);
```

Figure 5.2: A program that illustrates the NP-hardness of reasoning about assertions when the expression language uses combination of linear arithmetic and uninterpreted functions.

The above properties can be proved easily by induction on i. If $g \in \{0, \ldots, m-1\}$, then the assert statement is validated because:

$$\begin{aligned} h_{m-1} &= h_{m-1,g} \text{ (follows from property E2)} \\ &= h_{m-1,0} \text{ (follows from property E1)} \end{aligned}$$

If $g \notin \{0, \ldots, m-1\}$, then it follows from property E3 that the assert statement is invalidated. □

Lemma 15 implies the following theorem.

Theorem 17 *The assert statement in procedure* `IsUnSatisfiable`(ψ) *is true iff the input boolean formula* ψ *is unsatisfiable.*

Note that the two procedures `IsUnSatisfiable` and `Check` can be reduced to one procedure whose flowchart representation consists of only the nodes shown in Figure 5.1. (The program uses procedure calls and loops with deterministic predicates only for expository purposes.) This can be done by unrolling the loops and inlining procedure `Check` inside procedure `IsUnSatisfiable`. The size of the resulting procedure is polynomial in the size of the input boolean formula ψ.

5.2 Heuristic

In light of the hardness result in the previous section, we now discuss some heuristics to efficiently reason about the combined abstraction of linear arithmetic and unary uninterpreted functions. The techniques presented in this section can be used to reason about the combination of any two abstractions.

Note that the naive combination of the `SEval` functions for the individual expression languages is unsound. This is because the natural linear interpretation of linear arithmetic operators clashes with the random linear interpretation given to the uninterpreted functions. One way to resolve the above problem is to hash the value of an uninterpreted function term before being used in an arithmetic expression and vice versa. Such a hashing loses some information and prevents detection of some equal expressions, but it prevents the unintended interaction between the chosen linear interpretation of the uninterpreted function and the linear arithmetic operators. We first describe how to compute a mapping

T at each program point to keep track of the top-level operators used in computing the values of different variables. We then describe a `SEval` function for the combination of linear arithmetic and unary uninterpreted functions.

Top-level Operators

The mapping T specifies for each variable x, the kind of operator used in obtaining the value held by x. $T(x)$ is one of the following 4 values: $\bot$, `la`, `uf`, and $\top$. `la` and `uf` denote that the operator last used in obtaining the value for variable x was a linear arithmetic operator or an uninterpreted function respectively. $\bot$ denotes undefined while $\top$ denotes uncertainty. We compute the mapping T at each program point by performing a forward analysis on the flowchart nodes. The mapping T is updated for different flowchart nodes as described below, until a fixed point is reached (which requires a maximum of 3 iterations across any loop). Note that after a fixed point is reached, $T(x) \neq \bot$ for any variable x and mapping T at any program point.

Initialization: The mapping T at all program points except procedure entry is initialized to map every variable to $\bot$. At procedure entry, the mapping T is initialized to map every variable to $\top$.

Assignment Node: See Figure 5.1 (a).

The mapping T after the assignment node $x := e$ is obtained from the mapping T' before the assignment node as follows:

$$T = T'[x \leftarrow u]$$

where

$$u = \begin{cases} \texttt{la} & \text{if } e \text{ is } F(e_1) \\ \texttt{uf} & \text{if } e \text{ is } e_1 \pm e_2 \text{ or } c \\ T'[y] & \text{if } e \text{ is } y \end{cases}$$

Non-deterministic Assignment Node: See Figure 5.1 (b).

The mapping T after the assignment node is obtained from the mapping T' before the assignment node as follows:

$$T = T'[x \leftarrow \top]$$

Non-deterministic Conditional Node: See Figure 5.1 (c).
The mappings T^1 and T^2 on the two branches of the conditional node are simply copies of the mapping T before the conditional node.

$$T^1 = T^2 = T$$

Join Node: See Figure 5.1 (d).
The mapping T after the join node assigns to each variable x, the `Join` of the values assigned to x by the mappings T^1 and T^2 before the join node.

$$T(x) = \texttt{Join}(T^1(x), T^2(x))$$

The function `Join` is defined as follows:

`Join`	$\perp$	`la`	`uf`	$\top$
$\perp$	$\perp$	`la`	`uf`	$\top$
`la`	`la`	`la`	$\top$	$\top$
`uf`	`uf`	$\top$	`uf`	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

The `SEval` function

We propose the following `SEval` function for the random interpreter for combination of linear arithmetic and uninterpreted functions:

$$\begin{aligned}
&\texttt{SEval}(x) = x \\
&\texttt{SEval}(e) = \begin{cases} \texttt{SEval}_{\texttt{la}}(e) & \text{if } e \text{ is } e_1 \pm e_2 \text{ or } c \\ \texttt{SEval}_{\texttt{uf}}(e) & \text{if } e \text{ is } F(e_1) \end{cases}
\end{aligned}$$

The function $\texttt{SEval}_{\texttt{la}}$ uses the `SEval` function for linear arithmetic to interpret the linear arithmetic part of the expressions. For the sub-expressions that use uninterpreted function at the top level, it hashes the result returned by its counterpart $\texttt{SEval}_{\texttt{uf}}$.

$$\begin{aligned}
\texttt{SEval}_{\texttt{la}}(e_1 \pm e_2) &= \texttt{SEval}_{\texttt{la}}(e_1) \pm \texttt{SEval}_{\texttt{la}}(e_2) \\
\texttt{SEval}_{\texttt{la}}(c \times e) &= c \times \texttt{SEval}_{\texttt{la}}(e) \\
\texttt{SEval}_{\texttt{la}}(F(e_1, e_2)) &= \texttt{Hash}_{\texttt{la}}(\texttt{SEval}_{\texttt{uf}}(F(e_1, e_2))) \\
\texttt{SEval}_{\texttt{la}}(x) &= \texttt{HashIf}_{\texttt{uf}}(x)
\end{aligned}$$

The function $\texttt{SEval}_{\texttt{uf}}$ is similar to $\texttt{SEval}_{\texttt{la}}$ with its behavior swapped for the sub-expressions that use linear arithmetic and those that use uninterpreted function as the top-level operator.

$$\begin{aligned}
\texttt{SEval}_{\texttt{uf}}(F(e)) &= r_1 \times \texttt{SEval}_{\texttt{uf}}(e) + r_2 \\
\texttt{SEval}_{\texttt{uf}}(e_1 \pm e_2) &= \texttt{Hash}_{\texttt{uf}}(\texttt{SEval}_{\texttt{la}}(e_1 \pm e_2)) \\
\texttt{SEval}_{\texttt{uf}}(c \times e) &= \texttt{Hash}_{\texttt{uf}}(\texttt{SEval}_{\texttt{la}}(c \times e)) \\
\texttt{SEval}_{\texttt{uf}}(x) &= \texttt{HashIf}_{\texttt{la}}(x)
\end{aligned}$$

Note that the symbolic expressions defined by $\texttt{SEval}$ above uses new operations $\texttt{Hash}_{\texttt{la}}$, $\texttt{Hash}_{\texttt{uf}}$, $\texttt{HashIf}_{\texttt{la}}$, and $\texttt{HashIf}_{\texttt{uf}}$ apart from the regular addition and multiplication operations. The semantics of these new operations is as follows. For a given state ρ and a mapping $T(x)$, the $\texttt{Hash}_{\texttt{la}}$ and $\texttt{Hash}_{\texttt{uf}}$ functions simply hash the value of their arguments (i.e., they map value of their argument to a random integer from $\mathbb{F}_p$) while the functions $\texttt{HashIf}_{\texttt{la}}$ and $\texttt{HashIf}_{\texttt{uf}}$ either return their argument or its hash depending upon the value of $T(x)$:

$$\texttt{HashIf}_{\texttt{la}}(x) = \begin{cases} x & \text{if } T(x) = \texttt{uf} \\ \texttt{Hash}_{\texttt{la}}(x) & \text{if } T(x) = \texttt{la} \text{ or } \top \end{cases}$$

$$\texttt{HashIf}_{\texttt{uf}}(x) = \begin{cases} x & \text{if } T(x) = \texttt{la} \\ \texttt{Hash}_{\texttt{uf}}(x) & \text{if } T(x) = \texttt{uf} \text{ or } \top \end{cases}$$

The $\texttt{SEval}$ function defined above is sound (and hence the random interpretation scheme based on it is probabilistically sound). For example, consider the distinct expressions $e_1 = F(a+c) - F(b+c)$ and $e_2 = F(a) - F(b)$ introduced earlier in the chapter. Note that under the mapping $T(a) = T(b) = T(c) = \texttt{la}$, $\texttt{SEval}(e_1) = \texttt{Hash}_{\texttt{uf}}(r_1\texttt{Hash}_{\texttt{la}}(a+c) + r_2) - \texttt{Hash}_{\texttt{uf}}(r_1\texttt{Hash}_{\texttt{la}}(b+c)+r_2)$ and $\texttt{SEval}(e_2) = \texttt{Hash}_{\texttt{uf}}(r_1\texttt{Hash}_{\texttt{la}}(a)+r_2) - \texttt{Hash}_{\texttt{uf}}(r_1\texttt{Hash}_{\texttt{la}}(b)+r_2)$ and hence $\texttt{SEval}(e_1) \neq \texttt{SEval}(e_2)$. Similarly, it can be verified that under any mapping T, $\texttt{SEval}(e_1) \neq \texttt{SEval}(e_2)$.

The $\texttt{SEval}$ function defined above is not complete, i.e., it does not map equal expressions to equal values. For example, consider the two equal expressions $e_1 = F(1+a-1)$ and $e_2 = F(a)$. Note that $\texttt{SEval}(e_1) \neq \texttt{SEval}(e_2)$ since under the mapping $T(a) = \texttt{uf}$, $\texttt{SEval}(e_1) = r_1\texttt{Hash}_{\texttt{la}}(1 + \texttt{Hash}_{\texttt{uf}}(a) - 1) + r_2$, while $\texttt{SEval}(e_2) = r_1 a + r_2$. The precision of the $\texttt{SEval}$ function can be increased by ensuring that $\texttt{Hash}_{\texttt{uf}}$ and $\texttt{Hash}_{\texttt{la}}$ hash their arguments under the following constraint (where $\texttt{id}$ denotes the identity function).

$$\texttt{Hash}_{\texttt{uf}} \circ \texttt{Hash}_{\texttt{la}} = \texttt{id}$$
$$\texttt{Hash}_{\texttt{la}} \circ \texttt{Hash}_{\texttt{uf}} = \texttt{id}$$

The effect of the above constraint is to unhash a (seemingly) arithmetic value that is equal to an uninterpreted function value, when it is used inside an uninterpreted function term, and vice versa. This modification makes the $\texttt{SEval}$ function complete, i.e., it maps equal expressions to equal values (and hence the random interpretation scheme based on it can discover all equal expressions in a straight-line program). For example, note that $\texttt{SEval}(e_1) = \texttt{SEval}(e_2) = r_1 a + r_2$, for $e_1 = F(1 + a - 1)$ and $e_2 = F(a)$ under the mapping $T(a) = \texttt{uf}$. However, since the $\texttt{SEval}$ function is not linear (which violates property B4 in Section 4.1.1), the random interpretation scheme based on it remains incomplete for discovering equivalences in a program in presence of join points. This is not unexpected in view of the hardness result in Section 5.1, which implies that any polynomial time (and probabilistically sound) random interpretation for the combination of linear arithmetic and uninterpreted functions will be incomplete (unless RP=NP).

The intriguing possibility that axioms of randomness may constitute a useful fundamental source of mathematical truth independent of, but supplementary to, the standard axiomatic structure of mathematics suggests that probabilistic algorithms ought to be sought vigorously.

Jacob Schwartz, *Fast probabilistic algorithms for verification of polynomial identities.*

Chapter 6

Conclusion

A sound and complete program analysis is undecidable [Lan92]. A simple alternative is *random testing*, which is complete but unsound, in the sense that it cannot prove absence of bugs. At the other extreme, we have sound *abstract interpretation*, wherein we pay a price for the hardness of program analysis in terms of having an incomplete (i.e., conservative) analysis, or by having algorithms that are complicated and have long running-time. In this dissertation, we have described a new probabilistically sound program analysis technique called *random interpretation*, which can be simpler, more efficient, and more complete than its deterministic counterparts, at the price of degrading soundness from absolute certainty to guarantee with arbitrarily high probability.

Random interpretation can be regarded as a randomized version of abstract interpretation that uses random data-structures to represent invariants at each program point and randomized algorithms as its transfer functions. For example, in case of linear arithmetic, the subspace represented by linear equalities at each program point is represented by a few states chosen randomly from that subspace and the (randomized) affine join operation for performing join is more efficient than the traditional symbolic join operation. In case of uninterpreted functions, the Herbrand equivalences at each program point are represented by assigning hash values (which are random vectors) to different expressions such that (with high probability) two expressions have the same hash value iff they are equivalent.

Benefits of Random Interpretation

Randomization helps in making algorithms more efficient and precise at the cost of probabilistic soundness. This popular theme has been exploited earlier in several areas of computer science. In this dissertation, we have obtained similar benefits by applying randomization to program analysis problems. All the random interpretation based algorithms presented in this dissertation are more efficient than their deterministic counterparts, albeit at the cost of having a small error probability. However, this error probability can be made infinitesimally small by controlling some parameters of the algorithms so that for all practical purposes this error probability does not matter at all.

Another notable feature of random interpretation based algorithms is the simplicity of their data-structures and the operations that they perform on those data-structures. This is especially evident in the algorithms described in Chapter 2 and Chapter 3, where the data-structures maintained by the algorithm simply consists of mapping from program variables to integers, or tuple of integers.

An interesting aspect about randomization is that it inspires ideas for deterministic algorithms. We have described the first polynomial-time deterministic algorithm for the problem of global value numbering over the abstraction of uninterpreted functions and non-deterministic conditionals. The inspiration to develop a polynomial-time deterministic algorithm actually came after developing the randomized algorithm for this problem. We used several observations that were made while developing the randomized algorithm.

Proof Techniques for Random Interpretation

The most challenging aspect in this line of work has been proof of correctness (in particular, the proof of probabilistic soundness) of the algorithms. In each case, our intuition suggested that the probability of unsound results is extremely small, and experiments did not reveal any unsoundness. However, proving an upper bound for the probability of unsoundness was an extremely challenging task, and most often we had to settle with conservative bounds. One of the reasons for this difficulty was the lack of any proof techniques for proving correctness of randomized program analyses, simply because there were no such analyses known before. In the process of proving correctness of our algorithms, we have actually developed some new proof techniques, which might be useful for future random interpretation based program analyses.

The correctness proofs of analyses for the abstractions of linear arithmetic and uninterpreted functions share the theme of discovering an abstract interpreter that is as precise as the corresponding random interpreter, and then showing that the random interpreter simulates the actions of the abstract interpreter with high probability. The abstract interpreter, of course, is not as efficient as the random interpreter. For the linear arithmetic case, we show by induction that the error probability (the probability that the random interpreter deviates from the abstract interpreter) increases a tiny bit at each step, but still remains small at the end. For the uninterpreted functions case, we construct a symbolic version of the random interpreter and show that it simulates the abstract interpreter exactly. We then show that the random interpreter computes a random instantiation of the polynomials computed by the symbolic random interpreter, and the error probability is obtained by using the error bounds for the Schwartz and Zippel's polynomial identity testing algorithm.

The correctness proof of the inter-procedural analysis relies on constructing a fully-symbolic random interpreter whose correctness is not difficult to prove. We then prove by induction that the random interpreter computes exactly the same set of equivalences as the fully-symbolic random interpreter. This part is quite involved and requires developing some new concepts.

Scope of Random Interpretation

There appear to be two kinds of algorithmic challenges in program analysis. One of them involves developing analyses to reason about new abstractions, e.g., linear arithmetic, uninterpreted functions. The other kind of challenges involve developing techniques to improve the precision of given analyses. For example, lifting an intra-procedural analysis to inter-procedural setting, or combining different program analyses. Random Interpretation seems to be suited for addressing both kinds of challenges. Chapter 2 and Chapter 3 describe state-of-the-art algorithms, which are based on random interpretation, to reason about the abstractions of linear arithmetic and uninterpreted functions. Chapter 4 describes techniques to extend intra-procedural analyses to perform a precise inter-procedural reasoning, while Chapter 5 describes how to combine program analyses to do a more precise reasoning.

Combining randomization with symbolic algorithms can be quite a powerful technique. This is reflected in the symbolic random interpretation technique described in Chapter 4. Contrast this with the random interpretation algorithms described in Chapter 2 and Chapter 3, which resemble random testing procedures, from which they inherit trivial data structures and low complexity. The algorithms described in Chapter 4 start to mix randomization with symbolic analysis. The data structures become somewhat more involved, essentially consisting of random instances of otherwise symbolic data structures. Even the implementation of the algorithms starts to resemble that of symbolic deterministic algorithms. This change of style reflects our belief that the true future of randomization in program analysis is not in the form of a world parallel to traditional symbolic analysis algorithms, but in an organic mixture that exploits the benefits of both worlds.

Program analysis is provably hard, and we have all learned not to expect perfect results. However, this attitude has manifested itself mostly in a large number of static analysis approaches in which completeness is sacrificed and false positives are accepted as a fact of life, while soundness remains the *sine qua non* of program analysis. The results of this dissertation show that it might be profitable to relax this strict view of soundness, and trade off extremely small amount of soundness in return for other advantages such as better computational complexity, simplicity, or even more precise results. When we observe that in the grand scheme of things, program analyses are used to produce software that interacts with potentially buggy libraries running on fallible hardware, we realize that maybe a minuscule probability of unsoundness in the analysis is tolerable after all.

Bibliography

[AH87] L. Adleman and M. Huang. Recognizing primes in random polynomial time. In *19th Annual ACM Symposium on Theory of Computing*, pages 462–469, 1987.

[AKS02] M. Agrawal, N. Kayal, and N. Saxena. *Primes is in P.* Dept. of Computer Science and Engineering, Indian Institute of Technology, Kanpur, August 2002.

[AWZ88] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *15th Annual ACM Symposium on Principles of Programming Languages*, pages 1–11, 1988.

[BA98] Rastislav Bodik and Sadun Anik. Path-sensitive value-flow analysis. In *25th Annual ACM Symposium on Principles of Programming Languages*, pages 237–251, January 1998.

[BGLR93] M. Bellare, S. Goldwasser, C. Lund, and A. Russell. Efficient probabilistically checkable proofs and applications to approximation. In *25th Annual ACM Symposium on the Theory of Computing*, pages 294–304, May 1993.

[BW05] Andrej Bogdanov and Hoeteck Wee. More on noncommutative polynomial identity testing. In *20th Annual IEEE Conference on Computational Complexity.* IEEE, June 2005.

[Car02] Matthew C. Cary. Lattice basis reduction algorithms and applications. Unpublished survey paper. February 2002.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th Annual ACM Symposium on Principles of Programming Languages*, pages 234–252, 1977.

[CD89] Benny Chor and Cynthia Dwork. Randomization in Byzantine agreement. In *Advances in Computing Research*, volume 5, pages 443–497. JAI Press, 1989.

[CFR$^+$90] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1990.

[CH78] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *5th Annual ACM Symposium on Principles of Programming Languages*, pages 84–97, 1978.

[Cli95] Cliff Click. Global code motion/global value numbering. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 246–257, June 1995.

[FS98] Christian Fecht and Helmut Seidl. Propagating differences: an efficient new fixpoint algorithm for distributive constraint systems. In *Nordic Journal of Computing*, volume 5, pages 304–329, 1998.

[Gau86] C. F. Gauss. *Disquisitiones Arithmeticae.* Article 171, English Edition (Translated by A. Clarke). Springer-Verlag, New York, 1986.

[GN03] Sumit Gulwani and George C. Necula. Discovering affine equalities using random interpretation. In *30th Annual ACM Symposium on Principles of Programming Languages*, pages 74–84, January 2003.

[GN04a] Sumit Gulwani and George C. Necula. Global value numbering using random interpretation. In *31st Annual ACM Symposium on Principles of Programming Languages*, pages 342–352, January 2004.

[GN04b] Sumit Gulwani and George C. Necula. A polynomial-time algorithm for global value numbering. In *11th Static Analysis Symposium*, volume 3148 of *LNCS*, pages 212–227. Springer-Verlag, August 2004.

[GN05] Sumit Gulwani and George C. Necula. Precise interprocedural analysis using random interpretation. In *32nd Annual ACM Symposium on Principles of Programming Languages*, January 2005.

[Ham94] D. Hamlet. Random testing. In J. Marciniak, editor, *Encyclopedia of Software Engineering*, pages 970–978. Wiley, New York, 1994.

[Kar76] Michael Karr. Affine relationships among variables of a program. In *Acta Informatica*, pages 133–151. Springer, 1976.

[Kil73] G. A. Kildall. A unified approach to global program optimization. In *1st ACM Symposium on Principles of Programming Language*, pages 194–206. ACM, October 1973.

[Lan92] William Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, December 1992.

[MORS05] Markus Müller-Olm, Oliver Rüthing, and Helmut Seidl. Checking herbrand equalities and beyond. In *Verification, Model-Checking and Abstract Interpretation*, volume 3385 of *LNCS*, pages 79–96. Springer, January 2005.

[MOS04a] Markus Müller-Olm and Helmut Seidl. A note on Karr's algorithm. In *31st International Colloquium on Automata, Languages and Programming (ICALP)*, pages 1016–1028, 2004.

[MOS04b] Markus Müller-Olm and Helmut Seidl. Precise interprocedural analysis through linear algebra. In *31st Annual ACM Symposium on Principles of Programming Languages*, pages 330–341, January 2004.

[MOSS05] Markus Müller-Olm, Helmut Seidl, and Bernhard Steffen. Interprocedural herbrand equalities. In *Proceedings of the European Symposium on Programming*, volume 3444 of *LNCS*. Springer-Verlag, 2005.

[MR95] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[Muc00] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, 2000.

[Mul00] K. Mulmuley. Randomized algorithms in computational geometry. In J.R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, pages 703–724. Elsevier, 2000.

[Nec00] George C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 83–94, June 2000.

[NO79] Greg Nelson and Derek Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, October 1979.

[PRRR01] P. Pardalos, S. Rajasekaran, J. Reif, and J. Rolim, editors. *Handbook on Randomized Computing*. Kluwer Academic Publishers, Dordrecht, The Netherlands, June 2001.

[PSS98] Amir Pnueli, M. Siegel, and Eli Singerman. Translation validation. In Bernhard Steffen, editor, *4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, volume 1384 of *LNCS*, pages 151–166. Springer, 1998.

[Rep96] Thomas Reps. On the sequential nature of interprocedural program-analysis problems. *Acta Informatica*, 33(8):739–757, November 1996.

[RHS95] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *22nd Annual ACM Symposium on Principles of Programming Languages*, pages 49–61, 1995.

[RKS90] Oliver Rüthing, Jens Knoop, and Bernhard Steffen. The value flow graph: A program representation for optimal program transformations. In Neil D. Jones, editor, *Proceedings of the European Symposium on Programming*, volume 432 of *LNCS*, pages 389–405. Springer-Verlag, 1990.

[RKS99] Oliver Rüthing, Jens Knoop, and Bernhard Steffen. Detecting equalities of variables: Combining efficiency with precision. In *Static Analysis Symposium*, volume 1694 of *LNCS*, pages 232–247. Springer, 1999.

[RWZ88] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *15th ACM Symposium on Principles of Programming Languages*, pages 12–27, 1988.

[Sch80] J. T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *JACM*, 27(4):701–717, October 1980.

[SP81] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S.S. Muchnick and N.D. Jones, editors,. *Program Flow Analysis: Theory and Applications*, pages 189–234, 1981.

[SRH96] Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167(1–2):131–170, 30 October 1996.

[Ste87] Bernhard Steffen. Optimal run time optimization - proved by a new look at abstract interpretations. In *2nd International Joint Conference on Theory and Practice of Software Development (TAPSOFT)*, volume 249 of *LNCS*, pages 52–68. Springer-Verlag, March 1987.

[WZ91] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.

[Zip79] R. Zippel. Probabilistic algorithms for sparse polynomials. In *Proceedings of the International Symposium on Symbolic and Algebraic Manipulation (EUROSAM)*, volume 72 of *LNCS*, pages 216–226. Springer, June 1979.

Appendix A

Linear Arithmetic

We now prove the completeness theorem and soundness lemma stated in Section 2.2. Both the random interpreter and the abstract interpreter perform a forward analysis in the sense that the outputs of a flowchart node are determined by the inputs of that node. The proofs are by induction on the number of flowchart nodes analyzed by the interpreters. For the inductive case of the proof, we prove that the required property holds for the outputs of a node given that it holds for the inputs of that node. For the scenario when at least one of the inputs of a node is $\perp$ (undefined), the proof is trivial. Hence, we prove the results assuming that all inputs to any node are non-$\perp$.

A.1 Proof of Completeness (Theorem 1)

The proof is by induction on the number of nodes analyzed by the interpreters. For the base case, we have $U = \emptyset$. Hence, if $U \Rightarrow g = 0$, then it must be the case that g is identically equal to the 0 expression, and hence $S \models g = 0$. For the inductive case, the following scenarios arise.

Assignment Node: See Figure 2.6 (a).
Consider the expression $g' = g[e/x]$. Since $U \Rightarrow g = 0$, $U' \Rightarrow g' = 0$. It follows from the induction hypothesis on U' and S' that $S' \models g' = 0$. Hence, $S \models g = 0$.

Non-deterministic Assignment Node: See Figure 2.6 (b).
Since $U \Rightarrow g = 0$, it must be the case that the coefficient of x in expression g is 0, as the

set of linear equalities U does not involve any occurrence of variable x. Hence, $U' \Rightarrow g = 0$. It follows from the induction hypothesis on U' and S' that $S' \models g = 0$. Hence, $S \models g = 0$.

Conditional Node: See Figure 2.6 (c).
We prove that (a) $S^1 \models g = 0$, and (b) $S^2 \models g = 0$. Consider the following three possibilities.

- $U \Rightarrow e = 0$.
  It follows from the induction hypothesis on U and S that $S \models e = 0$. By definition of the abstract interpreter, $U^1 = U$ and $U^2 = \perp$. Similarly, $S^1 = S$ and $S^2 = \perp$.
  (a) Since $U^1 \Rightarrow g = 0$, we have that $U \Rightarrow g = 0$. It follows from the induction hypothesis on U and S that $S \models g = 0$. Thus, $S^1 \models g = 0$.
  (b) The proof obligation for S^2 is trivial.

- $U \Rightarrow e - c = 0$ for some non-zero constant c.
  It follows from the induction hypothesis on U and S that $S \models e - c = 0$. By definition, $U^1 = \perp$, $U^2 = U$, $S^1 = \perp$ and $S^2 = S$.
  (a) Since $U^2 \Rightarrow g = 0$, we have that $U \Rightarrow g = 0$. It follows from the induction hypothesis on U and S that $S \models g = 0$. Thus, $S^2 \models g = 0$.
  (b) The proof obligation for S^1 is trivial.

- $U \not\Rightarrow e - c = 0$ for any constant c.
  By definition, $U^1 = U \cup \{e = 0\}$, $U^2 = U$, $S^1 = \texttt{Adjust}(S, e)$, and $S^2 = S$.
  (a) Since $U^1 \models g = 0$, there exists an expression g' such that $U \Rightarrow g' = 0$ and $g = g' + \lambda e$ for some constant λ. It follows from induction hypothesis on U and S that $S \models g' = 0$. It follows from Lemma 3 that $S^1 \models g' = 0$ and $S^1 \models e = 0$. Hence, $S^1 \models g' + \lambda e = 0$.
  (b) Since $U^2 \Rightarrow g = 0$, we have that $U \Rightarrow g = 0$. It follows from induction hypothesis on U and S that $S \models g = 0$. Thus, $S^2 \models g = 0$.

Non-deterministic Conditional Node: See Figure 2.6 (d).
This case is trivial since $U^1 = U^2 = U$ and $S^1 = S^2 = S$.

Join Node: See Figure 2.6 (e).
Since $U \Rightarrow g = 0$, it follows from definition of the abstract interpreter that $U^1 \Rightarrow g = 0$

and $U^2 \Rightarrow g = 0$. By induction hypothesis on U^1 and S^1 and on U^2 and S^2, we have that $S^1 \models g = 0$ and $S^2 \models g = 0$. It now follows from Lemma 1 that $S \models g = 0$.

A.2 Proof of Soundness (Lemma 5)

The proof is by induction on the number of flowchart nodes analyzed by the interpreters. For the base case, we need to show the following since $q = 0$.

$$\Pr(\tilde{S} \models g = 0) \leq \left(\frac{1}{p}\right)^{\ell}$$

Note that $U = \emptyset$, and S is obtained by choosing random values for all variables in all of its states. Since $U \not\Rightarrow_p g = 0$, g is not identically equal to 0. The probability that some particular state in sample S satisfies the non-trivial linear equality $g = 0$ is at most $\frac{1}{p}$. Thus, the probability that some particular ℓ states from sample S satisfy the non-trivial linear equality $g = 0$ is at most $\left(\frac{1}{p}\right)^{\ell}$. For the inductive case, following scenarios arise.

Assignment Node: See Figure 2.6 (a).
Consider the expression $g' = g[e/x]$. Since $U \not\Rightarrow_p g = 0$, $U' \not\Rightarrow_p g' = 0$. Also, note that $S_i \models g = 0$ iff $S'_i \models g' = 0$. Let $\tilde{S}$ be some subset of ℓ states of sample S. Let $\tilde{S}'$ be the corresponding subset of sample S'. Let q and q' be the maximum number of adjust and join operations performed by the random interpreter on any path before computing samples S and S' respectively. Note that $q' = q$ and event $\mathcal{A}_{S'}$ is same as event $\mathcal{A}_S$. Hence,

$$\begin{aligned}\Pr(\tilde{S} \models g = 0 \parallel \mathcal{A}_S) &= \Pr(\tilde{S}' \models g' = 0 \parallel \mathcal{A}_{S'}) \\ &\leq \left(\frac{q'+1}{p}\right)^{\ell} \text{ (from induction hypothesis on } S')\end{aligned}$$

Non-deterministic Assignment Node: See Figure 2.6 (b).
Consider the following two cases:

- The coefficient of variable x in expression g is 0.
  Since $U \not\Rightarrow_p g = 0$, $U' \not\Rightarrow_p g = 0$. Also, note that $S_i \models g = 0$ iff $S'_i \models g = 0$. The result now follows easily from the induction hypothesis on U' and S'.

- The coefficient of variable x in expression g is not 0.
  Let $g = g' + \lambda x$ such that g' does not involve any occurrence of variable x. Note that

$\lambda \neq 0$. Let $\tilde{S} = \{S_{\sigma(1)}, \ldots, S_{\sigma(\ell)}\}$ be some subset of ℓ states of sample S. Then,

$$\begin{aligned}
\Pr\left(\tilde{S} \models g = 0 \parallel \mathcal{A}_S\right) &= \Pr\left(\bigwedge_{i=1}^{\ell} S_{\sigma(i)} \models g = 0 \parallel \mathcal{A}_S\right) \\
&= \Pr\left(\bigwedge_{i=1}^{\ell} \texttt{Eval}(g' + \lambda x, S_{\sigma(i)}) = 0 \parallel \mathcal{A}_S\right) \\
&= \Pr\left(\bigwedge_{i=1}^{\ell} \texttt{Rand}() = \frac{-\texttt{Eval}(g', S'_{\sigma(i)})}{\lambda} \parallel \mathcal{A}_S\right) \\
&= \bigwedge_{i=1}^{\ell} \Pr\left(\texttt{Rand}() = \frac{-\texttt{Eval}(g', S'_{\sigma(i)})}{\lambda}\right) \\
&= \left(\frac{1}{p}\right)^{\ell} \\
&\leq \left(\frac{q+1}{p}\right)^{\ell}
\end{aligned}$$

Conditional Node: See Figure 2.6 (c).

Let $\tilde{S^1}$ and $\tilde{S^2}$ be some subsets of ℓ states of samples S^1 and S^2 respectively. Let q, q_1 and q_2 be the maximum number of adjust and join operations performed by the random interpreter before computing samples S, S^1, and S^2 respectively on any path (from procedure entry to the corresponding program points). We prove that:

(a) $\Pr(\tilde{S^1} \models g = 0 \parallel \mathcal{A}_{S^1}) \leq \left(\frac{q_1+1}{p}\right)^{\ell}$

(b) $\Pr(\tilde{S^2} \models g = 0 \parallel \mathcal{A}_{S^2}) \leq \left(\frac{q_2+1}{p}\right)^{\ell}$

We first prove (b). Since $U^2 \not\models g = 0$, we have that $U^2 \neq \perp$. This implies that $U \not\Rightarrow_p e = 0$ and $U^2 = U$. If the sample S is sound, then $S \not\Rightarrow e = 0$, and hence $S^2 = S$. The result now follows trivially from the induction hypothesis on U and S.

We now prove (a). Since $U^1 \not\models g = 0$, we have that $U^1 \neq \perp$. This implies that $U \not\Rightarrow_p e = c$ for any non-zero constant c. Suppose that the sample S is sound. Then, $S \not\models e = c$ for any non-zero constant c. If $S \models e = 0$, then $S^1 = S$, $q_1 = q$, $U^1 = U$, and the result follows easily from the induction hypothesis on U and S. We now consider the case when $S \not\models e = 0$. In that case $S^1 = \texttt{Adjust}(S, e)$, $q_1 = q + 1$, and $U^1 = U \cup \{e = 0\}$. Let $\tilde{S}$ be the subset of ℓ states of sample S corresponding to the subset $\tilde{S^1}$ of S^1, i.e., $\tilde{S} = \{S_i \parallel S_i^1 \in \tilde{S^1}\}$. Let $\tilde{S'} = \tilde{S} \cup \{S_{j_1}^1, S_{j_2}^1\}$, where j_1 and j_2 are as defined in the `Adjust` operation. Let $\mathcal{B}$ denote the event that $\texttt{Eval}(e, S_{j_1}) \neq \texttt{Eval}(e, S_{j_2})$. Let $\mathcal{G}$ be the event that

$S_{j_1} \models g + ce = 0 \wedge S_{j_2} \models g + ce = 0$ for any constant $c \in \mathbb{F}_p$.

We first consider the case when $\tilde{S^1}$ does not include $S^1_{j_1}$ and $S^1_{j_2}$.

$$\begin{aligned}
&\Pr(\tilde{S^1} \models g = 0 \parallel \mathcal{A}_{S^1})\\
&\le \sum_{c\in\mathbb{F}_p} \Pr(\tilde{S} \models g + ce = 0 \wedge \rho_0 \models g + ce = 0 \parallel \mathcal{A}_{S^1} \wedge \mathcal{B}) \text{ (follows from Lemma 4)}\\
&= \sum_{c\in\mathbb{F}_p} \Pr(\tilde{S} \models g + ce = 0 \parallel \mathcal{A}_{S^1} \wedge \mathcal{B}) \times \Pr(\rho_0 \models g + ce = 0 \parallel \tilde{S} \models g + ce = 0 \wedge \mathcal{A}_{S^1} \wedge \mathcal{B})\\
&\le \sum_{c\in\mathbb{F}_p} \Pr(\tilde{S} \models g + ce = 0 \parallel \mathcal{A}_{S^1} \wedge \mathcal{B}) \times (\Pr(\mathcal{G} \parallel \tilde{S} \models g + ce = 0 \wedge \mathcal{A}_{S^1} \wedge \mathcal{B})\\
&\quad + \Pr(\rho_0 \models g + ce = 0 \parallel \neg\mathcal{G} \wedge \tilde{S} \models g + ce = 0 \wedge \mathcal{A}_{S^1} \wedge \mathcal{B}))\\
&= \sum_{c\in\mathbb{F}_p} (\Pr(\tilde{S}' \models g + ce = 0 \parallel \mathcal{A}_{S^1} \wedge \mathcal{B})\\
&\quad + \Pr(\tilde{S} \models g + ce = 0 \parallel \mathcal{A}_{S^1} \wedge \mathcal{B}) \times \Pr(\rho_0 \models g + ce = 0 \parallel \neg\mathcal{G}))\\
&\le \sum_{c\in\mathbb{F}_p} (\Pr(\tilde{S}' \models g + ce = 0 \parallel \mathcal{A}_{S^1} \wedge \mathcal{B}) + \Pr(\tilde{S} \models g + ce = 0 \parallel \mathcal{A}_{S^1} \wedge \mathcal{B}) \times \frac{1}{p-(t+1)}) \quad \text{(A.1)}
\end{aligned}$$

We now lower bound $\Pr(\tilde{S}' \models g + ce = 0 \parallel \mathcal{A}_{S^1} \wedge \mathcal{B})$ and $\Pr(\tilde{S} \models g + ce = 0 \parallel \mathcal{A}_{S^1} \wedge \mathcal{B})$. Let $\mathcal{A}$ denote the event that sample S is sound.

$$\begin{aligned}
\Pr(\tilde{S}' \models g + ce = 0 \parallel \mathcal{A}_{S^1} \wedge \mathcal{B}) &= \Pr(\tilde{S}' \models g + ce = 0 \parallel \mathcal{A}_S \wedge \mathcal{A} \wedge \mathcal{B})\\
&= \frac{\Pr(\tilde{S}' \models g + ce = 0 \wedge \mathcal{A}_S \wedge \mathcal{A} \wedge \mathcal{B})}{\Pr(\mathcal{A}_S \wedge \mathcal{A} \wedge \mathcal{B})}\\
&\le \frac{\Pr(\tilde{S}' \models g + ce = 0 \wedge \mathcal{A}_S)}{\Pr(\mathcal{A}_S \wedge \mathcal{A} \wedge \mathcal{B})}\\
&= \frac{\Pr(\tilde{S}' \models g + ce = 0 \wedge \mathcal{A}_S)}{\Pr(\mathcal{A}_S)} \times \frac{\Pr(\mathcal{A}_S)}{\Pr(\mathcal{A}_S \wedge \mathcal{A} \wedge \mathcal{B})}\\
&= \frac{\Pr(\tilde{S}' \models g + ce = 0 \parallel \mathcal{A}_S)}{\Pr(\mathcal{A} \wedge \mathcal{B} \parallel \mathcal{A}_S)} \quad \text{(A.2)}
\end{aligned}$$

Similarly,

$$\Pr(\tilde{S} \models g + ce = 0 \parallel \mathcal{A}_{S^1} \wedge \mathcal{B}) \le \frac{\Pr(\tilde{S} \models g + ce = 0 \parallel \mathcal{A}_S)}{\Pr(\mathcal{A} \wedge \mathcal{B} \parallel \mathcal{A}_S)} \quad \text{(A.3)}$$

We now lower bound $\Pr(\mathcal{A} \wedge \mathcal{B} \parallel \mathcal{A}_S)$. Note that

$$\begin{aligned}
\Pr(\neg\mathcal{A} \parallel \mathcal{A}_S) &\leq \sum_{g' \text{ s.t. } U \not\Rightarrow_p g'=0} \Pr(S \models g' = 0 \parallel \mathcal{A}_S) \\
&\leq \sum_{g' \text{ s.t. } U \not\Rightarrow_p g'=0} \left(\frac{q+1}{p}\right)^t \text{ (from induction hypothesis on } S) \\
&\leq \frac{p^{k+1}-p}{p-1} \times \left(\frac{q+1}{p}\right)^t \\
&\leq \frac{1}{p^2} \text{ (assuming } t \geq 3k/2+3 \text{ and } p \geq (q+1)^3+1) \qquad \text{(A.4)}
\end{aligned}$$

Also, note that

$$\begin{aligned}
\Pr(\neg\mathcal{B} \parallel \mathcal{A}_S) &\leq \sum_{c' \in \mathbb{F}_p} \Pr(S_{j_1} \models e = c' \wedge S_{j_2} \models e = c' \parallel \mathcal{A}_S) \\
&\leq \sum_{c' \in \mathbb{F}_p} \left(\frac{q+1}{p}\right)^2 \text{ (from induction hypothesis on } S) \\
&= p \times \left(\frac{q+1}{p}\right)^2 = \frac{(q+1)^2}{p} \qquad \text{(A.5)}
\end{aligned}$$

Thus,

$$\begin{aligned}
\Pr(\mathcal{A} \wedge \mathcal{B} \parallel \mathcal{A}_S) &= 1 - \Pr(\neg\mathcal{A} \vee \neg\mathcal{B} \parallel \mathcal{A}_S) \\
&\geq 1 - \Pr(\neg\mathcal{A} \parallel \mathcal{A}_S) - \Pr(\neg\mathcal{B} \parallel \mathcal{A}_S) \\
&\geq 1 - \frac{1}{p^2} - \frac{(q+1)^2}{p} \text{ (from Eq. A.4 and A.5)} \\
&= \frac{p - \left(\frac{1}{p} + (q+1)^2\right)}{p} \qquad \text{(A.6)}
\end{aligned}$$

Using the induction hypothesis on S and Eq. A.6 in Eq. A.2 and A.3, we obtain bounds on $\Pr(\tilde{S}' \models g + ce = 0 \parallel \mathcal{A}_{S^1} \wedge \mathcal{B})$ and $\Pr(\tilde{S} \models g + ce = 0 \parallel \mathcal{A}_{S^1} \wedge \mathcal{B})$ as follows:

$$\Pr(\tilde{S}' \models g + ce = 0 \parallel \mathcal{A}_{S^1} \wedge \mathcal{B}) \leq \left(\frac{q+1}{p}\right)^{\ell+2} \times \frac{p}{p - \left(\frac{1}{p} + (q+1)^2\right)} \qquad \text{(A.7)}$$

$$\Pr(\tilde{S} \models g + ce = 0 \parallel \mathcal{A}_{S^1} \wedge \mathcal{B}) \leq \left(\frac{q+1}{p}\right)^{\ell} \times \frac{p}{p - \left(\frac{1}{p} + (q+1)^2\right)} \qquad \text{(A.8)}$$

Using Eq. A.7 and A.8 in Eq. A.1, we obtain:

$$\begin{aligned}
&\Pr(\tilde{S}^1 \models g = 0 \parallel \mathcal{A}_{S^1}) \\
&\leq \sum_{c \in \mathbb{F}_p} \left(\frac{q+1}{p}\right)^\ell \times \frac{p}{p - \left(\frac{1}{p} + (q+1)^2\right)} \times \left(\left(\frac{q+1}{p}\right)^2 + \frac{1}{p-(t+1)}\right) \\
&= p \times \left(\frac{q+1}{p}\right)^\ell \times \frac{p}{p - \left(\frac{1}{p} + (q+1)^2\right)} \times \left(\left(\frac{q+1}{p}\right)^2 + \frac{1}{p-(t+1)}\right) \\
&= \left(\frac{q+1}{p}\right)^\ell \times \frac{p}{p - \left(\frac{1}{p} + (q+1)^2\right)} \times \left(\frac{(q+1)^2}{p} + \frac{p}{p-(t+1)}\right) \\
&\leq \left(\frac{q+1}{p}\right)^\ell \times \frac{p}{p - \left(\frac{1}{p} + (q+1)^2\right)} \times \left(\frac{(q+1)^2 + p}{p-(t+1)}\right) \\
&\leq \left(\frac{q+1}{p}\right)^\ell \times \frac{p}{p - \left(\frac{1}{p} + (q+1)^2\right)} \times \frac{p}{p - ((q+1)^2 + t + 1)} \\
&\leq \left(\frac{q+1}{p}\right)^\ell \times \frac{p}{p - \left(\frac{1}{p} + 2(q+1)^2 + t + 1\right)} \\
&\leq \left(\frac{q+2}{p}\right)^\ell \text{ (assuming } p \geq (q+2)(2(q+1)^2 + t + 2)) \\
&= \left(\frac{q_1+1}{p}\right)^\ell \qquad \text{(A.9)}
\end{aligned}$$

We now consider the case when $\tilde{S}^1$ includes $S^1_{j_1}$ or $S^2_{j_2}$. Let $\tilde{S}^{1,1} = \tilde{S}^1 - \{S^1_{j_1}, S^1_{j_2}\}$. Let $\tilde{S}^{1,2} = \tilde{S}^1 - \tilde{S}^{1,1}$. The states in $\tilde{S}^{1,2}$ are spread u.a.r. in the subspace defined by the states in $\tilde{S^1}$. If sample S is sound, then it follows from Lemma 4 that sample S^1 is sound, and hence $S^1 \not\models g = 0$. Thus, we have

$$\Pr(\tilde{S}^{1,2} \models g = 0 \parallel \tilde{S^{1,1}} \models g = 0 \wedge \mathcal{A}_{S^1}) \leq \left(\frac{1}{p}\right)^{|\tilde{S}^{1,2}|} \qquad \text{(A.10)}$$

Hence,

$$\begin{aligned}
&\Pr(\tilde{S}^1 \models g = 0 \parallel \mathcal{A}_{S^1}) \\
&= \Pr(\tilde{S}^{1,1} \models g = 0 \wedge \tilde{S}^{1,2} \models g = 0 \parallel \mathcal{A}_{S^1}) \\
&= \Pr(\tilde{S}^{1,1} \models g = 0 \parallel \mathcal{A}_{S^1}) \times \Pr(\tilde{S}^{1,2} \models g = 0 \parallel \tilde{S^{1,1}} \models g = 0 \wedge \mathcal{A}_{S^1}) \\
&\leq \left(\frac{q_1+1}{p}\right)^{|\tilde{S}^{1,1}|} \times \left(\frac{1}{p}\right)^{|\tilde{S}^{1,2}|} \text{ (from Eq. A.9 and A.10)} \\
&\leq \left(\frac{q_1+1}{p}\right)^{|\tilde{S}^{1,1}| + |\tilde{S}^{1,2}|} = \left(\frac{q_1+1}{p}\right)^\ell
\end{aligned}$$

Non-deterministic Conditional Node: See Figure 2.6 (d).
This case is trivial since $U^1 = U^2 = U$ and $S^1 = S^2 = S$.

Join Node: See Figure 2.6 (e).
Let q_1 and q_2 be the maximum number of adjust and join operations performed by the random interpreter before computing samples S^1 and S^2 respectively on any path (from procedure entry to the corresponding program points). Clearly, $q = 1 + max(q_1, q_2)$. Since $U \not\Rightarrow_p g = 0$, either $U^1 \not\Rightarrow_p g = 0$ or $U^2 \not\Rightarrow_p g = 0$. Consider the case when $U^1 \not\Rightarrow_p g = 0$. (The other case is symmetric.) Let $\tilde{S}$ be any subset of ℓ states of sample S. Let $I(\tilde{S}) = \{i \parallel S_i \in \tilde{S}\}$. For $I \subseteq I(\tilde{S})$, let $\mathcal{E}_I$ be the event that $\texttt{Eval}(g, S_i^1) = 0$ iff $i \in I$. The events $\mathcal{E}_I$ form a disjoint partition of the event space. Let $\mathcal{F}_i$ be the event that $\texttt{Eval}(g, S_i^1) \neq \texttt{Eval}(g, S_i^2)$. Let $c_i = \frac{\texttt{Eval}(g,S_i^2)}{\texttt{Eval}(g,S_i^2) - \texttt{Eval}(g,S_i^1)}$. Let $S^{1,I} = \{S_i^1 \parallel i \in I\}$. Then,

$$\begin{aligned}
&\Pr(\tilde{S} \models g = 0 \parallel \mathcal{A}_S) \\
&= \sum_{I \subseteq I(\tilde{S})} \Pr(\mathcal{E}_I \parallel \mathcal{A}_S) \times \Pr(\tilde{S} \models g = 0 \parallel \mathcal{E}_I \wedge \mathcal{A}_S) \\
&\leq \sum_{I \subseteq I(\tilde{S})} \Pr(S^{1,I} \models g = 0 \parallel \mathcal{A}_S) \times \Pr(\tilde{S} \models g = 0 \parallel \mathcal{E}_I \wedge \mathcal{A}_S) \\
&= \sum_{I \subseteq I(\tilde{S})} \Pr(S^{1,I} \models g = 0 \parallel \mathcal{A}_{S^1}) \times \Pr(\bigwedge_{i \in I(\tilde{S}) - I} \mathcal{F}_i \wedge \texttt{Rand}() = c_i \parallel \mathcal{E}_I \wedge \mathcal{A}_S) \\
&\leq \sum_{I \subseteq I(\tilde{S})} \Pr(S^{1,I} \models g = 0 \parallel \mathcal{A}_{S^1}) \times \bigwedge_{i \in I(\tilde{S}) - I} \Pr(\texttt{Rand}() = c_i \parallel \mathcal{F}_i) \\
&\leq \sum_{I \subseteq I(\tilde{S})} \left(\frac{q_1 + 1}{p}\right)^{|I|} \times \left(\frac{1}{p}\right)^{\ell - |I|} \quad \text{(from induction hypothesis on } S^1\text{)} \\
&= \sum_{i=1}^{\ell} \binom{\ell}{i} \left(\frac{q_1 + 1}{p}\right)^{|I|} \times \left(\frac{1}{p}\right)^{\ell - |I|} \\
&= \left(\frac{q_1 + 2}{p}\right)^{\ell} \\
&= \left(\frac{q + 1}{p}\right)^{\ell}
\end{aligned}$$

Appendix B

Uninterpreted Functions

The actions of the interpreters for a non-deterministic assignment $x :=?$ are same as for an assignment $x := x'$, where x' is some fresh variable (can be thought of as a new input variable). Hence, the non-deterministic assignment can be regarded as a special case of the deterministic assignment. Therefore, we omit the case of a non-deterministic assignment node in all inductive proofs in this chapter. Note that this reduction does not lead to any increase in any parameter on which the computation complexity of the algorithm depends.

We first prove the completeness and soundness theorems stated in Section 3.2. Both the random interpreter and the abstract interpreter perform a forward analysis in the sense that the outputs of a flowchart node are determined by the inputs of that node. The proofs are by induction on the number of flowchart nodes analyzed by the interpreters. For the inductive case of the proof, we prove that the required property holds for the outputs of a node given that it holds for the inputs of that node. For the scenario when at least one of the inputs of a node is $\perp$ (undefined), the proof is trivial. Hence, we prove the results assuming that all inputs to any node are non-$\perp$.

B.1 Proof of Completeness (Theorem 8)

The proof is by induction on the number of flowchart nodes analyzed by the interpreters. The base case is trivial since initially $U = \emptyset$. Since $\emptyset \Rightarrow e_1 = e_2$, it must be the case that $e_1 \equiv e_2$. Hence, $\rho \models e_1 = e_2$. For the inductive case, the following scenarios arise.

Assignment Node: See Figure 3.4 (a).
Consider the expressions $e'_1 = e_1[e/x]$ and $e'_2 = e_2[e/x]$. Since $U \Rightarrow e_1 = e_2$, we have that $U' \Rightarrow e'_1 = e'_2$. It follows from the induction hypothesis on U' and $\tilde{\rho}'$ that $\tilde{\rho}' \models e'_1 = e'_2$. Hence, $\tilde{\rho} \models e_1 = e_2$.

Non-deterministic Conditional Node: See Figure 3.4 (c).
This case is trivial since $U^1 = U^2 = U$ and $\tilde{\rho}^1 = \tilde{\rho}^2 = \tilde{\rho}$. By using the induction hypothesis on $\tilde{\rho}$ and U, we get the desired result.

Join Node: See Figure 3.4 (d).
By definition of the abstract interpreter, $U^1 \Rightarrow e_1 = e_2$ and $U^2 \Rightarrow e_1 = e_2$. By induction hypothesis on U^1 and $\tilde{\rho}^1$ and on U^2 and $\tilde{\rho}^2$, we have that $\tilde{\rho}^1 \models e_1 = e_2$ and $\tilde{\rho}^2 \models e_1 = e_2$. It now follows from Lemma 8 that $\tilde{\rho} \models e_1 = e_2$.

B.2 Proof of Soundness (Theorem 9)

The proof is by induction on the number of flowchart nodes analyzed by the interpreters. We first prove the base case. Note that $A(e_1, \tilde{\rho}) = \texttt{SEval}(e_1)$ and $A(e_2, \tilde{\rho}) = \texttt{SEval}(e_2)$ since $\tilde{\rho}(x) = [x, \ldots, x]^T$. Hence, $\texttt{SEval}(e_1) = \texttt{SEval}(e_2)$. The result now follows from Lemma 7. For the inductive case, the following scenarios arise.

Assignment Node: See Figure 3.4 (a).
Consider the expressions $e'_1 = e_1[e/x]$ and $e'_2 = e_2[e/x]$. Note that $\tilde{\rho}' \models e'_1 = e'_2$ since $\tilde{\rho} \models e_1 = e_2$. Also, $\mathit{Degree}(A(e_1, \tilde{\rho})) = \mathit{Degree}(A(e'_1, \tilde{\rho}'))$. Hence, $\ell \geq \mathit{Degree}(A(e'_1, \tilde{\rho}'))$ since $\ell \geq \mathit{Degree}(A(e_1, \tilde{\rho}))$. It follows from the induction hypothesis on U' and $\tilde{\rho}'$ that $U' \Rightarrow e'_1 = e'_2$. Thus, it follows that $U \Rightarrow e_1 = e_2$.

Non-deterministic Conditional Node: See Figure 3.4 (b).
This case is trivial since $\tilde{\rho}^1 = \tilde{\rho}^2 = \tilde{\rho}$ and $U^1 = U^2 = U$. The induction hypothesis on $\tilde{\rho}$ and U implies the desired result.

Join Node: See Figure 3.4 (c).
By definition of $\tilde{\rho}$, $A(e_1, \tilde{\rho}) = w \times A(e_1, \tilde{\rho}^1) + (1-w) \times A(e_1, \tilde{\rho}^2)$ and $A(e_2, \tilde{\rho}) = w \times A(e_2, \tilde{\rho}^1) + (1-w) \times A(e_2, \tilde{\rho}^2)$, where w is a fresh variable that does not occur in $A(e_1, \tilde{\rho}^1)$, $A(e_1, \tilde{\rho}^2)$,

$A(e_2, \tilde{\rho}^1)$, or $A(e_2, \tilde{\rho}^2)$. Since $A(e_1, \tilde{\rho}) = A(e_2, \tilde{\rho})$, it follows that $A(e_1, \tilde{\rho}^1) = A(e_2, \tilde{\rho}^1)$ (by substituting $w = 0$). Hence, $\tilde{\rho}^1 \models e_1 = e_2$. Also, $\mathit{Degree}(A(e_1, \tilde{\rho}^1)) = \mathit{Degree}(A(e_1, \tilde{\rho}))$ and $\mathit{Degree}(A(e_2, \tilde{\rho}^1)) = \mathit{Degree}(A(e_2, \tilde{\rho}))$. Thus, it follows from the induction hypothesis on U^1 and $\tilde{\rho}^1$ that $U^1 \Rightarrow e_1 = e_2$. Similarly, we can prove that $U^2 \Rightarrow e_1 = e_2$. It now follows from the definition of the abstract interpreter that $U \Rightarrow e_1 = e_2$.

B.3 Proof of Lemma 9

We first introduce some useful notation. We say that a pair $H = (V, E)$ has property $\mathcal{P}$ if $V \subseteq V_\pi$ and E is a set of equivalences $x = e$, one for each variable $x \in V_\pi - V$, such that $\mathit{Vars}(e) \subseteq V$. Furthermore, if $x \in V$, then $x \prec y$ for all variables y such that $(y = x) \in E$.

For any pair $H = (V, E)$ with property $\mathcal{P}$, let $\prec_E$ denote a partial order on the program variables such that $y \prec_E z$ iff $E \Rightarrow z = F(e_1, e_2)$ and $y \in \mathit{Vars}(F(e_1, e_2))$, or $E \Rightarrow y = z$ such that $y \prec z$.

For any equivalence $e_1 = e_2$, let $D_{e_1=e_2}$ denote the following set of equivalences.

$$\begin{aligned} D_{F(e_1',e_2')=F(e_1'',e_2'')} &= D_{e_1'=e_1''} \cup D_{e_2'=e_2''} \\ D_{y=y} &= \{\} \\ D_{y=e} &= \{y = e\},\ \mathit{where}\ e \not\equiv y \end{aligned}$$

Note that $D_{e_1=e_2}$ contains only equivalences of the form $y = e$, where y is some variable and e is some expression. Observe that

$$E \Rightarrow e_1 = e_2 \text{ iff } E \Rightarrow D_{e_1=e_2}$$

(By $E \Rightarrow D_{e_1=e_2}$, we mean that $E \Rightarrow e = e'$ for every equivalence $(e = e')$ in $D_{e_1=e_2}$.)

We now prove the lemma. The proof of the lemma is by induction on structure of the program. The base case is trivial since the pair $H = (V_\pi, \emptyset)$ clearly has property $\mathcal{P}$ and represents the set of Herbrand equivalences at procedure entry. For the inductive case, the following scenarios arise.

Assignment Node: See Figure 3.4 (a).

Let $H' = (V', E')$ represent the set of Herbrand equivalences before the assignment node such that H' has property $\mathcal{P}$. We define below a pair $H = (V, E)$ that has property $\mathcal{P}$ and that represents the set of Herbrand equivalences after the assignment node.

$$\begin{aligned}
E_t &= E'[x'/x] \cup \{x = Sub(e, E')[x'/x]\} \\
V &= \{y \mid y \in V_\pi, \neg\exists e'(E_t \Rightarrow y = e',\ e' \prec y,\ x' \notin Vars(e'))\} \\
E &= \{y = e' \mid y \in V_\pi - V,\ E_t \Rightarrow y = e', e' \prec y, Vars(e') \subseteq V\}
\end{aligned}$$

Here $Sub(e, E')$ refers to the expression obtained from e by replacing all occurrences of variable y in e by e' for all equivalences $y = e'$ in E'. Note that x' represents a fresh variable that does not occur in V_π.

It is not difficult to see that $H = (V, E)$ has property $\mathcal{P}$. Clearly, E_t represents the set of Herbrand equivalences after the assignment node, since it is the strongest post-condition of E with respect to the assignment $x := e$. Hence, it suffices to show that if $E_t \Rightarrow e_1 = e_2$ such that $x' \notin Vars(e_1)$ and $x' \notin Vars(e_2)$, then $E \Rightarrow e_1 = e_2$. Suppose $E_t \Rightarrow e_1 = e_2$. Then, $E_t \Rightarrow D_{e_1=e_2}$. It now follows from Claim 1 stated and proved below that $E \Rightarrow D_{e_1=e_2}$. Hence, $E \Rightarrow e_1 = e_2$.

Claim 1 *Suppose $E_t \Rightarrow y = e'$ such that $x' \notin Vars(e')$ and the variable y is different from variable x'. Then $E \Rightarrow y = e'$.*

Proof. Without loss of generality, we can assume that $e' \prec y$. The proof is by induction on the ordering of variable y in the partial order $\prec_E$. The base case corresponds to y being in the set V, and hence $e' \equiv y$. Clearly, $E \Rightarrow y = y$. We now consider the inductive case. Since $E_t \Rightarrow y = e'$, there exists an expression e'' such that $(y = e'') \in E$. Note that $E_t \Rightarrow y = e''$ and hence $E_t \Rightarrow e' = e''$. Thus, $E_t \Rightarrow D_{e'=e''}$. Note that for every $(z = e''') \in D_{e'=e''}$, $z \prec_E y$; and hence, it follows from the inductive hypothesis that $E \Rightarrow z = e'''$. Thus, $E \Rightarrow D_{e'=e''}$, and hence $E \Rightarrow e' = e''$. Thus, $E \Rightarrow y = e'$. $\square$

Non-deterministic Conditional Node: See Figure 3.4 (c).
This case is trivial since the sets of Herbrand equivalences that are true on the two branches are same as the set of Herbrand equivalences that are true before the conditional.

Join Node: See Figure 3.4 (d).
Let $H_1 = (V_1, E_1)$ and $H_2 = (V_2, E_2)$ represent the set of Herbrand equivalences before the

join node such that both H_1 and H_2 have property $\mathcal{P}$. We define below a pair $H = (V, E)$ that has property $\mathcal{P}$ and that represents the set of Herbrand equivalences after the join node.

$$V = \{y \mid y \in V_\pi, \neg\exists e'(E_1 \Rightarrow y = e',\ E_2 \Rightarrow y = e', e' \prec y,)\}$$
$$E = \{y = e' \mid y \in V_\pi - V,\ E_1 \Rightarrow y = e',\ E_2 \Rightarrow y = e', e' \prec y,\ \mathit{Vars}(e') \subseteq V\}$$

It is not difficult to see that $H = (V, E)$ has property $\mathcal{P}$. It suffices to show that if $E_1 \Rightarrow e_1 = e_2$ and $E_2 \Rightarrow e_1 = e_2$, then $E \Rightarrow e_1 = e_2$. Suppose $E_1 \Rightarrow e_1 = e_2$ and $E_2 \Rightarrow e_1 = e_2$. Then, $E_1 \Rightarrow D_{e_1=e_2}$ and $E_2 \Rightarrow D_{e_1=e_2}$. It now follows from Claim 2 stated and proved below that $E \Rightarrow D_{e_1=e_2}$. Hence, $E \Rightarrow e_1 = e_2$.

Claim 2 *Suppose $E_1 \Rightarrow y = e'$ and $E_2 \Rightarrow y = e'$. Then $E \Rightarrow y = e'$.*

Proof. Without loss of generality, we can assume that $e' \prec y$. The proof is by induction on the ordering of variable y in the partial order $\prec_{E_1}$. The base case corresponds to y being in the set V_1, and hence $e' \equiv y$. Clearly, $E \Rightarrow y = y$. We now consider the inductive case. Since $E_1 \Rightarrow y = e'$ and $E_2 \Rightarrow y = e'$, there exists an expression e'' such that $(y = e'') \in E$. Since $E \Rightarrow y = e''$, we have that $E_1 \Rightarrow y = e''$ and $E_2 \Rightarrow y = e''$. Hence, $E_1 \Rightarrow e' = e''$ and $E_2 \Rightarrow e' = e''$. Thus, $E_1 \Rightarrow D_{e'=e''}$ and $E_2 \Rightarrow D_{e'=e''}$. For every $(z = e''') \in D_{e'=e''}$, $z \preceq_{E_1} y$; and hence, it follows from the inductive hypothesis that $E \Rightarrow (z = e''')$. Thus, $E \Rightarrow D_{e'=e''}$, and hence $E \Rightarrow e' = e''$. Thus, $E \Rightarrow y = e'$. $\square$

Appendix C

Inter-procedural Analysis

C.1 Definition and Properties of Fully Symbolic State

Phase 1

We hypothetically extend the random interpreter to also compute a symbolic state $\tilde{\rho}$, paths T, and an integer d (also referred to as d_S) at each program point besides a sample S. Paths T represent the set of all paths analyzed by the random interpreter inside the corresponding procedure. A path is simply a sequence of assignments. The symbolic state $\tilde{\rho}$ gives for each variable x that polynomial whose different random instantiations are the values of x in different states in the sample S. The integer d represents the maximum degree of the weight variables (introduced at join nodes and procedure call nodes) in any polynomial in the symbolic state $\tilde{\rho}$. d also represents the maximum number of join points and procedure calls analyzed by the random interpreter along any path immediately after computation of sample S. The values of $\tilde{\rho}$, T and d are updated for each flowchart node as shown below.

Initialization: Following initialization is done at procedure entry points.

$$\tilde{\rho}(x) = \texttt{SEval}(x)$$

$$T = \{\epsilon\}$$

$$d = 0$$

ϵ denotes the empty sequence of assignment nodes. At all other program points, the following initialization is done: $\tilde{\rho} = \perp$, $T = \emptyset$, and $d = -1$.

Assignment Node: See Figure 4.4 (a).
If $\tilde{\rho}' = \bot$, then $\tilde{\rho} = \bot$, $T = \emptyset$ and $d = -1$. Else,

$$\begin{aligned} \tilde{\rho} &= \tilde{\rho}'[x \leftarrow (\texttt{SEval}(e))[\tilde{\rho}']] \\ T &= \{\tau; x := e \parallel \tau \in T'\} \\ d &= d' \end{aligned}$$

$(\texttt{SEval}(e))[\tilde{\rho}']$ refers to the polynomial obtained from $\texttt{SEval}(e)$ by replacing all variables y by $\tilde{\rho}'(y)$.

Non-deterministic Assignment Node: See Figure 4.4 (b).
If $\tilde{\rho}' = \bot$, then $\tilde{\rho} = \bot$, $T = \emptyset$ and $d = -1$. Else,

$$\begin{aligned} \tilde{\rho} &= \tilde{\rho}'[x \leftarrow x'], \text{ where } x' \text{ is a fresh variable} \\ T &= \{\tau; x :=? \parallel \tau \in T'\} \\ d &= d' \end{aligned}$$

Non-deterministic Conditional Node: See Figure 4.4 (c).

$$\begin{aligned} \tilde{\rho}^1 &= \tilde{\rho}^2 = \tilde{\rho} \\ T^1 &= T^2 = T \\ d^1 &= d^2 = d \end{aligned}$$

Join Node: See Figure 4.4 (d).
If $\tilde{\rho}^1 = \bot$, then $\tilde{\rho} = \tilde{\rho}^2$, $T = T^2$ and $d = d^2$. Else if $\tilde{\rho}^2 = \bot$, then $\tilde{\rho} = \tilde{\rho}^1$, $T = T^1$ and $d = d^1$. Else,

$$\begin{aligned} \tilde{\rho}(x) &= \alpha\tilde{\rho}^1(x) + (1-\alpha)\tilde{\rho}^2(x), \text{ for all variables } x \\ T &= T^1 \cup T^2 \\ d &= max(d^1, d^2) + 1 \end{aligned}$$

α is a fresh variable that does not occur in $\tilde{\rho}^1$ and $\tilde{\rho}^2$.

Procedure Call: See Figure 4.4 (e).

If $\tilde{\rho}' = \perp$ or $Y_{P'} = \perp$, then $\tilde{\rho} = \perp$, $T = \emptyset$ and $d = -1$. Else,

$$\tilde{\rho}(x) = \begin{cases} \tilde{Y}_{P'}(x)[\tilde{\rho}'(y_1)/y_1, .., \tilde{\rho}'(y_k)/y_k] & \text{if } x \in O_{P'} \\ \tilde{\rho}'(x) & \text{otherwise} \end{cases}$$

$$T = \{\tau_1; \tau_2 \parallel \tau_1 \in T', \tau_2 \in \mathit{Paths}(Y_{P'})\}$$

$$d = d' + 1$$

where,

$$\tilde{Y}_{P'} = \sum_{j=1}^{t-1} \alpha_j Y_{P',j} + (1 - \sum_{j=1}^{t-1} \alpha_j) Y_{P',t}$$

α_j's are fresh variables that do not occur in $\tilde{\rho}'$ and $Y_{P',j}$. $O_{P'}$ is the set of output variables of procedure P' and $y_1, \ldots, y_k$ are the input variables of procedure P'. $\mathit{Paths}(Y_P)$ refers to be the set of paths T after the exit node of procedure P during computation of the set of runs Y_P for procedure P. Initially, Y_P is defined to be $\perp$ and $\mathit{Paths}(Y_P)$ is defined to be the empty set for all procedures P.

We say that a summary Y_P is sound and complete for a context C when $Y_P \models_C e_1 = e_2 \iff \mathit{Holds}(e_1 = e_2, \mathit{Paths}(Y_P), C)$.

Lemma 16 *Suppose that the summaries of all procedures plugged into analyzing a procedure P are sound and complete for all contexts. Let $\tilde{\rho}$ be the fully-symbolic state and T be the set of paths computed by the random interpreter at any program point inside procedure P (in phase 1). Let C be any context for procedure P and $e_1 = e_2$ be some equivalence. Then,*

$$\tilde{\rho} \models_C e_1 = e_2 \iff \mathit{Holds}(e_1 = e_2, T, C)$$

Proof. The proof of the lemma is by induction on the number of flowchart nodes analyzed by the random interpreter. The base case follows easily from the soundness and completeness properties (properties B1 and B2) of the `SEval` function. For the inductive case, the proof is trivial if one of the inputs of a node is $\perp$; hence we consider the scenarios when all inputs to a node are non-$\perp$.

Assignment Node: See Figure 4.4 (a).

Let $e_1' = e_1[e/x]$ and $e_2' = e_2[e/x]$. Note that $e_1' = e_2'$ is the weakest precondition of $e_1 = e_2$

immediately before the assignment node along paths in T. Hence,

$$\mathit{Holds}(e_1 = e_2, T, C) \iff \mathit{Holds}(e_1' = e_2', T', C)$$

It follows from the induction hypothesis on $\tilde{\rho}'$ that

$$\mathit{Holds}(e_1' = e_2', T', C) \iff \tilde{\rho}' \models_C e_1' = e_2'$$

It now follows from property B3 of the `SEval` function that

$$\tilde{\rho}' \models_C e_1' = e_2' \iff \tilde{\rho} \models_C e_1 = e_2$$

Non-deterministic Assignment Node: See Figure 4.4 (b).
Let x' be the fresh variable assigned to x by the symbolic random interpreter in obtaining state $\tilde{\rho}$ from $\tilde{\rho}'$. Let $e_1' = e_1[x'/x]$ and $e_2' = e_2[x'/x]$. Note that $e_1' = e_2'$ is the weakest precondition of $e_1 = e_2$ along paths in T immediately before the assignment node. The rest of the proof is now similar to the case of assignment node above.

Non-deterministic Conditional Node: See Figure 4.4(c).
This case is trivial since $\tilde{\rho}^1 = \tilde{\rho}$ and $\tilde{\rho}^2 = \tilde{\rho}$.

Join Node: See Figure 4.4(d).
Note that

$$\begin{aligned}\mathit{Holds}(e_1 = e_2, T, C) \iff & \mathit{Holds}(e_1 = e_2, T^1, C) \\ & \text{and } \mathit{Holds}(e_1 = e_2, T^2, C)\end{aligned}$$

It follows from the induction hypothesis on $\tilde{\rho}^1$ and on $\tilde{\rho}^2$ that

$$\begin{aligned}\mathit{Holds}(e_1 = e_2, T^1, C) &\iff \tilde{\rho}^1 \models_C e_1 = e_2 \\ \mathit{Holds}(e_1 = e_2, T^2, C) &\iff \tilde{\rho}^2 \models_C e_1 = e_2\end{aligned}$$

Since $\tilde{\rho}' = \alpha\tilde{\rho}^1 + (1-\alpha)\tilde{\rho}^2$, the following holds:

$$\tilde{\rho}' \models_C e_1 = e_2 \iff \tilde{\rho}^1 \models_C e_1 = e_2 \text{ and } \tilde{\rho}^2 \models_C e_1 = e_2$$

Procedure Call: See Figure 4.4 (e).

Let $\tilde{\rho}_j$ be the following symbolic state:

$$\tilde{\rho}_j(x) = \begin{cases} Y_{P',j}(x)[\tilde{\rho}'(y_1)/y_1, .., \tilde{\rho}'(y_k)/y_k] & \text{if } x \in O_{P'} \\ \tilde{\rho}'(x) & \text{otherwise} \end{cases}$$

where $O_{P'}$ is the set of output variables of procedure P' and $y_1, \ldots, y_k$ are the input variables of procedure P'. It follows from the induction hypothesis on $\tilde{\rho}'$ and the soundness and completeness of the summary for procedure P' that

$$\mathit{Holds}(e_1 = e_2, T, C) \iff \forall j \in \{1, .., t\},\ \tilde{\rho}_j \models_C e_1 = e_2$$

Since $\tilde{\rho}' = \sum\limits_{j=1}^{t-1} \alpha_j \tilde{\rho}_j + (1 - \sum\limits_{j=1}^{t-1} \alpha_j)\tilde{\rho}_t$, the following holds:

$$\tilde{\rho} \models_C e_1 = e_2 \iff \forall j \in \{1, .., t\},\ \tilde{\rho}_j \models_C e_1 = e_2$$

□

Phase 2

We hypothetically extend the random interpreter to also compute a symbolic state $\tilde{\rho}$, paths T, and an integer d (also referred to as d_S) at each program point besides a sample S, as is done in proving the correctness of phase 1. These are updated for different flowchart nodes as in phase 1, except for the initialization of any procedure other than `Main`.

The entry point of any procedure P other than `Main` is initialized as follows. Let there be m call sites for procedure P that have a non-$\perp$ sample. Let $T^i, d^i, \tilde{\rho}^i$ be the values computed by the random interpreter at the i^{th} such call site. Then, the random interpreter performs the following initialization for the entry point of procedure P.

$$\begin{aligned} T &= T^1 \cup .. \cup T^m \\ \tilde{\rho} &= \sum_{i=1}^{m-1} \alpha_i \tilde{\rho}^i + (1 - \sum_{i=1}^{m-1} \alpha_i)\tilde{\rho}^m \\ d &= max(d^1, .., d^m) + 1 \end{aligned}$$

Here $\alpha_1, \ldots, \alpha_{m-1}$ are fresh variables.

We use the notation $\mathit{Holds}_2(e_1 = e_2, T)$ to denote that the equivalence $e_1 = e_2$ holds at the end of all paths in T. We also use the notation $\rho \models e_1 = e_2$ to denote that $\texttt{Eval}(e_1, \rho) = \texttt{Eval}(e_2, \rho)$ for any state ρ.

Lemma 17 *Suppose that the summaries of all procedures plugged into analyzing a procedure P are sound and complete for all contexts. Let $\tilde{\rho}$ be the fully-symbolic state and T be the set of paths computed by the random interpreter at any program point inside procedure P (in phase 2). Let $e_1 = e_2$ be any equivalence. Then,*

$$\tilde{\rho} \models e_1 = e_2 \iff \mathit{Holds}_2(e_1 = e_2, T)$$

Proof. The proof is by induction on the number of flowchart nodes analyzed by the random interpreter. The base case follows easily from the soundness and completeness of the `SEval` function. For the inductive case, the proofs for assignment node (both deterministic and non-deterministic), non-deterministic conditional node, join node, and procedure call are similar to the ones for phase 1. We now prove the inductive case for the entry point of a procedure P. Let $\pi^1, .., \pi^m$ be the program points immediately before the calls to procedure P that have a non-$\perp$ sample. Let $\tilde{\rho}^i, T^i, d^i$ be the values computed by the random interpreter at those points. The following holds:

$$\mathit{Holds}_2(e_1 = e_2, T) \iff \forall i \in \{1, .., m\}, \mathit{Holds}_2(e_1 = e_2, T^i)$$

It follows from the induction hypothesis on $\tilde{\rho}^i$ that

$$\mathit{Holds}_2(e_1 = e_2, T^i) \iff \tilde{\rho}^i \models e_1 = e_2$$

Since $\tilde{\rho} = \sum_{i=1}^{m-1} \alpha_i \tilde{\rho}^i + (1 - \sum_{j=1}^{m-1})\alpha_m \tilde{\rho}^m$, the following holds:

$$\tilde{\rho} \models e_1 = e_2 \iff \forall i \in \{1, .., m\}\ \tilde{\rho}^i \models e_1 = e_2$$

□

C.2 Equivalent program without any procedure calls

In this section, we show how to convert each procedure in a program (in our program model) into an equivalent procedure (in the sense that both procedures satisfy the same set of equivalences at corresponding program points) that does not use any procedure calls. It follows from Theorem 15 and Theorem 16 that any program node is processed at most H_1 times in phase 1 and H_2 times in phase 2 during fixed-point computation. We use this observation to transform the given program (with procedure calls) in the following manner:

1. In the original program, unroll all loops inside procedures and in the call graph along the paths taken by any standard summary based inter-procedural analyzer in phase 1. It follows from Theorem 15 that this unrolling leads to an H_1 times increase in the size of the procedures.

2. In the acyclic call graph with acyclic procedures obtained in step 1, replace all procedure calls by recursive procedure inlining in a bottom-up manner.

The procedures thus obtained have at most m^m nodes, where m is the size of the program obtained in step 1 (measured in terms of the number of nodes). Thus, the size of each procedure is bounded above by $(nH_1)^{nH_1}$ nodes.

Next, observe that there exists a generalized context (which is an acyclic program fragment) for each procedure P in the sense that whatever equivalences hold among the input variables of P in all calls to P, the same set of equivalences hold among those variables at the end of the generalized context. The generalized context can be obtained as follows:

3. In the original program, unroll all loops in the call graph and inside procedures along the paths taken by any standard summary based inter-procedural analyzer in phase 2. It follows from Theorem 16 that this unrolling increases the size of the procedures by a factor of at most H_2.

4. In the acyclic call graph with acyclic procedures thus obtained in step 3, build a context (in a bottom-up manner) for a procedure P, which has q call sites inside procedures $P_1, \ldots, P_q$, as follows. Construct a non-deterministic conditional with q branches, whose i^{th} branch is the context of procedure P_i followed by the code of P_i that leads up to the corresponding call to procedure P.

The contexts thus obtained have a worst-case size of m^m, where m is the size of the program obtained in step 3. This leads to a total size of $(nH_2)^{nH_2}$ for each context.

We can now obtain the desired program without any procedure calls as follows:

5. In the original program, prepend all procedures by their generalized contexts obtained in step 4. This leads to a total size of at most $2(nH_2)^{nH_2}$ nodes for each procedure.

6. In the program obtained in step 5, replace all procedure calls by their summaries obtained in step 2. This leads to a total size of at most $n_{max} = 2(nH_1)^{nH_1}(nH_2)^{nH_2}$ nodes for each procedure.

C.3 Proof of Lemma 13

Let S be any sample and $\tilde{\rho}$ be the corresponding fully-symbolic state of k variables computed by the random interpreter at some program point. For any state ρ of k variables, let $J(\rho)$ denote the vector $(\rho(x_1), \ldots, \rho(x_k), 1)$. Let A be the set of elements of the vectors in the set $\{J(\tilde{\rho}')[v_i/w_i] \parallel v_i \in \mathbb{F}_p\}$, where $J(\tilde{\rho}')[v_i/w_i]$ denotes the vector obtained from $J(\tilde{\rho}')$ by replacing all weight variables w_i by some choices of elements v_i from $\mathbb{F}_p$. Let $\mathbb{F}'_p$ be the smallest field generated by the elements in set A. Let $\mathcal{U}(\tilde{\rho}')$ be the vector space generated by the vectors $\{J(\tilde{\rho}')[v_i/w_i] \mid v_i \in \mathbb{F}_p\}$ over the field $\mathbb{F}'_p$. Let m be the rank of this vector space. Note that $m \le 1 + k$ (since there can be at most $1 + k$ linearly independent vectors over $\mathbb{F}'_p$, where each vector consists of $1 + k$ elements from $\mathbb{F}'_p$).

Let $\mathcal{E}$ be the event that the vectors $J(S'_1), \ldots, J(S'_t)$ have less than m linearly independent vectors over the field $\mathbb{F}'_p$. We partition the event $\mathcal{E}$ into disjoint cases depending on which of the vectors $J(S'_1), \ldots, J(S'_t)$ are linearly independent. Let I be any subset of $\{1, .., t\}$. Let $\mathcal{F}_i$ be the event that $J(S'_i)$ is linearly independent of $J(S'_1), \ldots, J(S'_{i-1})$. Let $\mathcal{E}_I$ be the event $\bigwedge\limits_{i \in I} \mathcal{F}_i \wedge \bigwedge\limits_{i \in \{1,\ldots,t\}-I} \neg\mathcal{F}_i$. The set of events $\{\mathcal{E}_I \mid I \subseteq \{1, .., t\}, 1 \in I, |I| < m\}$ is a disjoint partition of the probability space for event $\mathcal{E}$. Thus,

$$\Pr(\mathcal{E}) = \sum_{I \subseteq \{1,..,t\}, 1 \in I, |I| < m} \Pr(\mathcal{E}_I)$$

$$\begin{aligned}
\Pr(\mathcal{E}_I) &= \Pr(\bigwedge_{i \in I} \mathcal{F}_i \quad \wedge \bigwedge_{i \in \{1,..,t\}-I} \neg\mathcal{F}_i) \\
&= \prod_{i \in I} \Pr(\mathcal{F}_i \mid \bigwedge_{j \in I, j<i} \mathcal{F}_j \quad \wedge \bigwedge_{j \in \{1,..,t\}-I, j<i} \neg\mathcal{F}_j] \\
&\quad \times \prod_{i \in \{1,..,t\}-I} \Pr(\neg\mathcal{F}_i \mid \bigwedge_{j \in I, j<i} \mathcal{F}_j \quad \wedge \bigwedge_{j \in \{1,..,t\}-I, j<i} \neg\mathcal{F}_j) \\
&\le \prod_{i \in \{1,..,t\}-I} \Pr(\neg\mathcal{F}_i \mid \bigwedge_{j \in I, j<i} \mathcal{F}_j \quad \wedge \bigwedge_{j \in \{1,..,t\}-I, j<i} \neg\mathcal{F}_j)
\end{aligned}$$

We now bound $\Pr(\neg\mathcal{F}_i \mid \bigwedge\limits_{j \in I, j<i} \mathcal{F}_j \quad \wedge \bigwedge\limits_{j \in \{1,..,t\}-I, j<i} \neg\mathcal{F}_j)$ for any $i \in \{1, .., t\} - I$. Let I_i be the set $\{j \in I \mid j < i\}$ and let $n_i = |I_i|$. Let M_i be the matrix with $n_i + 1$ rows from the set $\{J(S'_j) \parallel j \in I_i \cup \{i\}\}$. Let $\tilde{M}_i$ be the matrix with $n_i + 1$ rows obtained from M_i by replacing the row $J(S'_i)$ with $J(\tilde{\rho}')$. Since the events $\{\mathcal{F}_j\}_{j \in I_i}$ occur, the vectors $\{J(S'_j)\}_{j \in I_i}$ are linearly independent. Note that $J(\tilde{\rho}')$ is linearly independent of the vectors

$\{J(S'_j)\}_{j \in I_i}$ (because otherwise $Rank(\mathcal{U}(\tilde{\rho}'))$ would be n_i, which is less than m). Hence, $Rank(\tilde{M}_i) = n_i + 1$. Thus, there exists a submatrix $\tilde{M}_i^s$ of $\tilde{M}_i$ of size $(n_i + 1) \times (n_i + 1)$ such that $Rank(\tilde{M}_i^s) = n_i + 1$, or equivalently, $Det(\tilde{M}_1^s) \not\equiv 0$. Let M_i^s be the submatrix of M_i consisting of those columns of M_i that are used in obtaining $\tilde{M}_i^s$ from $\tilde{M}_i$. Note that S'_i is a random instantiation of $\tilde{\rho}'$, and hence $Det(M_i^s)$ is a random instantiation [1] of $Det(\tilde{M}_i^s)$, which is identically not equal to 0. Hence, it follows from the classic theorem on checking polynomial identities [Sch80, Zip79] that $\Pr(Det(M_i^s) = 0) \leq \frac{D}{p}$, where D is the degree of polynomial $Det(\tilde{M}_i^s)$ (in the variables whose random instantiation is used to obtain S'_i from $\tilde{\rho}'$). Note that $D \leq d_S$. Since $J(S'_i)$ is linearly dependent on $\{J(S'_j)\}_{j \in I_i}$ only if $Det(M_1^s) = 0$, we have:

$$\Pr(\neg \mathcal{F}_i \mid \bigwedge_{j \in I, j<i} \mathcal{F}_j \ \wedge \bigwedge_{j \in \{1,..,t\}-I, j<i} \neg \mathcal{F}_j) \leq \frac{d_S}{p}$$

Thus,

$$\begin{aligned}
\Pr(\mathcal{E}_I) &\leq \prod_{i \in \{1,..,t\}-I} \Pr(\neg \mathcal{F}_i \mid \bigwedge_{j \in I, j<i} \mathcal{F}_j \ \wedge \bigwedge_{j \in \{1,..,t\}-I, j<i} \neg \mathcal{F}_j) \\
&\leq \prod_{i \in \{1,..,t\}-I} \frac{d_S}{p} \ = \ \left(\frac{d_S}{p}\right)^{t-|I|}
\end{aligned}$$

$$\begin{aligned}
\Pr(\mathcal{E}) &= \sum_{I \subseteq \{1,..,t\}, 1 \in I, |I|<m} \Pr(\mathcal{E}_I) \\
&\leq \sum_{I \subseteq \{1,..,t\}, 1 \in I, |I|<m} \left(\frac{d_S}{p}\right)^{t-|I|} \\
&\leq \sum_{i=1}^{m-1} \binom{t-1}{i-1} \left(\frac{d_S}{p}\right)^{t-i} \\
&\leq \sum_{i=1}^{m-1} \left(\frac{3(t-1)}{t-i}\right)^{t-i} \times \left(\frac{d_S}{p}\right)^{t-i} \\
&\leq \sum_{i=1}^{m-1} \left(\frac{3t}{t-(m-1)} \times \frac{d_S}{p}\right)^{t-i} \\
&\leq \frac{\alpha^{t-k_{\mathtt{v}}}}{1-\alpha}, \text{ where } \alpha = \frac{3 d_S t}{p(t-k_{\mathtt{v}})}
\end{aligned}$$

[1]This makes use of the assumption that `SEval` function does not involve any random variables because otherwise the choice of the random variables in S'_i is already decided by the choices that occur in other states S'_j and hence in $\mathtt{Det}(\tilde{M}_i^s)$.

We now show that $\gamma'_1(S) \leq \Pr(\mathcal{E})$. Suppose that event $\mathcal{E}$ does not occur. Consider some equivalence $e_1 = e_2$ that is not entailed by $\tilde{\rho}$ in some context C. We show that S also does not entail the equivalence $e_1 = e_2$ in context C. Let $e = \texttt{SEval}(e_1) - \texttt{SEval}(e_2)$. Since $\tilde{\rho}$ does not entail $e_1 = e_2$ in context C, there exists some concrete state ρ belonging to the vector space $\mathcal{U}(\tilde{\rho})$ such that ρ does not entail $e_1 = e_2$ in context C, or equivalently, $e[\rho][C] \neq 0$. Since event $\mathcal{E}$ does not occur, there exist $\alpha_1, \ldots, \alpha_t \in \mathbb{F}'_p$ such that $J(\rho) = \sum_{i=1}^{t} \alpha_i J(S_i)$. Since $\sum_{i=1}^{t} \alpha_i = 1$, we have $e[\rho][C] = \sum_{i=1}^{t} (\alpha_i e[S_i])[C] = \sum_{i=1}^{t} \alpha_i[C](e[S_i][C])$. This implies that there exists $1 \leq i \leq t$ such that $e[S_i][C] \neq 0$. Thus, S_i (and hence S) does not entail the equivalence $e_1 = e_2$ in context C. This completes the proof.

C.4 Proof of Lemma 14

We first define the notion of a *basic* set of contexts.

Definition 2 [Basic Set of Contexts] A set of contexts B for a procedure P is said to be *basic* when for all contexts C and all equivalences $e_1 = e_2$ (such that the variables that occur in e_1 and e_2 have mappings in C), if $\texttt{Eval}(e_1, C) \neq \texttt{Eval}(e_2, C)$ then there exists a context $C' \in B$ such that $\texttt{Eval}(e_1, C') \neq \texttt{Eval}(e_2, C')$ and $C' \Rightarrow C$. We denote such a context C' by $\mathit{Basic}_B(C, e_1 = e_2)$.

A basic set of contexts has the following property.

Property 5 *Let B be a basic set of contexts for a procedure P. Suppose that a summary Y_P for a procedure P is sound for all contexts in B. Then Y_P is sound for all contexts for procedure P.*

Proof. Let C be any context and $e_1 = e_2$ be any equivalence such that

$$\neg(\mathit{Holds}(e_1 = e_2, \mathit{Paths}(Y_P), C))$$

This implies that there exists a path $\tau \in \mathit{Paths}(Y_P)$ such that

$$\texttt{Eval}(e_1^\tau, C) \neq \texttt{Eval}(e_2^\tau, C)$$

where $e_i^\tau = e_i[g_m/x_m]..[g_1/x_1]$, and τ is the sequence of assignments $x_1 = g_1; \ldots; x_m = g_m$. Let C_τ be $\mathit{Basic}_B(C, e_1^\tau = e_2^\tau)$. By definition of C_τ, we have

$$\texttt{Eval}(e_1^\tau, C_\tau) \neq \texttt{Eval}(e_2^\tau, C_\tau)$$

and hence

$$\neg(\mathit{Holds}(e_1 = e_2, \mathit{Paths}(Y_P), C_\tau))$$

It now follows from the soundness of Y_P for context C_τ that

$$\neg(Y_P \models_{C_\tau} e_1 = e_2)$$

Since $C_\tau \Rightarrow C$, we conclude that

$$\neg(Y_P \models_C e_1 = e_2)$$

$\square$

Observe that the following set is a basic set of contexts for any procedure P.

$$B = \{\{y_1 = v_1, .., y_k = v_k\} \parallel y_i \in I_P, c_i \in \mathbb{F}_p\} \text{ where } v_i \equiv \texttt{SEval}(y_i)[c_i/y_i]$$

I_P denotes the set of input variables of procedure P. Note that $|B| = p^{k_\mathtt{i}}$. Hence, $N \leq p^{k_\mathtt{i}}$.