Program Analysis

Jonathan Aldrich, Claire Le Goues, and Rohan Padhye

(a work in progress; last updated on March 31, 2021)

Spring 2021

Contents

Chapter 1

Introduction

Software is transforming the way that we live and work. We communicate with friends via social media on smartphones, and use websites to buy what we need and to learn about anything in the world. At work, software helps us organize our businesses, reach customers, and distinguish ourselves from competitors.

Unfortunately, it is still challenging to produce high-quality software, and much of the software we do use has bugs and security vulnerabilities. Recent examples of problems caused by buggy software include uncontrollable acceleration in Toyota cars and a glitch in Nest smart thermostats left many homes without heat. Just looking at one category of defect, software race conditions, we observe problems ranging from power outages affecting millions of people in the US Northeast in 2003 to deadly radiation overdoses from the Therac-25 radiation therapy machine.

Program analysis is all about analyzing software code to learn about its properties. Program analyses can find bugs or security vulnerabilities like the ones mentioned above. It can also be used to synthesize test cases for software, and even to automatically patch software. For example, Facebook uses the Getafix tool to automatically produce patches for bugs found by other analysis tools.[1] Finally, program analysis is used in compiler optimizations in order to make programs run faster.

This book covers both foundations and practical aspects of the automated analysis of programs, which is becoming increasingly critical to find software errors, assure program correctness, and discover properties of code. We start by looking at how we can use mathematical formalisms to reason about how programs execute, then examine how programs are represented within compilers and analysis tools. We study dataflow analysis and the corresponding theory of abstract interpretation, which captures the essence of a broad range of program analyses and supports reasoning about their correctness. Building on this foundation, later chapters will describe various kinds of dataflow analysis, pointer analysis, interprocedural analysis, and symbolic execution.

In course assignments that go with this book, students will design and implement analysis tools that find bugs and verify properties of software. Students will apply knowledge and skills learned in the course to a capstone research project that involves designing, implementing, and evaluating a novel program analysis.

Overall program analysis is an area with deep mathematical foundations that is also very practically useful. I hope you will also find it to be fun!

[1] See https://code.fb.com/developer-tools/getafix-how-facebook-tools-learn-to-fix-bugs-automatically/

Chapter 2

The WHILE Language and Program Representation

2.1 The WHILE Language

We will begin our study of the theory of analyses using a simple programming language called WHILE, with various extensions. The WHILE language is at least as old as Hoare's 1969 paper on a logic for proving program properties. It is a simple imperative language, with (to start!) assignment to local variables, if statements, while loops, and simple integer and boolean expressions.

We use the following metavariables to describe different categories of syntax. The letter on the left will be used as a variable representing a piece of a program. On the right, we describe the kind of program piece that variable represents:

S	statements
a	arithmetic expressions (AExp)
x, y	program variables (Vars)
n	number literals
b	boolean expressions (BExp)

The syntax of WHILE is shown below. Statements S can be an assignment $x := a$; a skip statement, which does nothing;[1] and `if` and `while` statements, with boolean expressions b as conditions. Arithmetic expressions a include variables x, numbers n, and one of several arithmetic operators (op_a). Boolean expressions include true, false, the negation of another Boolean expression, Boolean operators op_b applied to other Boolean expressions, and relational operators op_r applied to arithmetic expressions.

$$
\begin{array}{rcl@{\qquad}rcl@{\qquad}rcl@{\qquad}rcl}
S & ::= & x := a & b & ::= & \text{true} & a & ::= & x & op_b & ::= & \text{and} \mid \text{or} \\
 & \mid & \text{skip} & & \mid & \text{false} & & \mid & n & op_r & ::= & < \mid \leqslant \mid = \\
 & \mid & S_1;\ S_2 & & \mid & \text{not } b & & \mid & a_1\ op_a\ a_2 & & \mid & > \mid \geqslant \\
 & \mid & \text{if } b \text{ then } S_1 \text{ else } S_2 & & \mid & b_1\ op_b\ b_2 & & & & op_a & ::= & + \mid - \mid * \mid / \\
 & \mid & \text{while } b \text{ do } S & & \mid & a_1\ op_r\ a_2 & & & &
\end{array}
$$

[1] Similar to a lone semicolon or open/close bracket in C or Java

2.2 WHILE3ADDR: A Representation for Analysis

For analysis, the source-like definition of WHILE can sometimes prove inconvenient. For example, WHILE has three separate syntactic forms—statements, arithmetic expressions, and boolean predicates—and we would have to define the semantics and analysis of each separately to reason about it. A simpler and more regular representation of programs will help simplify certain of our formalisms.

As a starting point, we will eliminate recursive arithmetic and boolean expressions and replace them with simple atomic statement forms, which are called *instructions*, after the assembly language instructions that they resemble. For example, an assignment statement of the form $w = x * y + z$ will be rewritten as a multiply instruction followed by an add instruction. The multiply assigns to a temporary variable t_1, which is then used in the subsequent add:

$$t_1 = x * y$$
$$w = t_1 + z$$

As the translation from expressions to instructions suggests, program analysis is typically studied using a representation of programs that is not only simpler, but also lower-level than the source (WHILE, in this instance) language. Many Java analyses are actually conducted on byte code, for example. Typically, high-level languages come with features that are numerous and complex, but can be reduced into a smaller set of simpler primitives. Working at the lower level of abstraction thus also supports simplicity in the compiler.

Control flow constructs such as `if` and `while` are similarly translated into simpler jump and conditional branch constructs that jump to a particular (numbered) instruction. For example, a statement of the form `if` b `then` S_1 `else` S_2 would be translated into:

$$\begin{array}{ll} 1: & \text{if } b \text{ then goto } 4 \\ 2: & S_2 \\ 3: & \text{goto } 5 \\ 4: & S_1 \end{array}$$

Exercise 1. How would you translate a WHILE statement of the form `while` b `do` S?

This form of code is often called 3-address code, because every instruction has at most two source operands and one result operand. We now define the syntax for 3-address code produced from the WHILE language, which we will call WHILE3ADDR. This language consists of a set of simple instructions that load a constant into a variable, copy from one variable to another, compute the value of a variable from two others, or jump (possibly conditionally) to a new address n. A program P is just a map from addresses to instructions:[2]

$$\begin{array}{rcll} I & ::= & x := n & \\ & | & x := y & \\ & | & x := y \; op \; z & \\ & | & \text{goto } n & \\ & | & \text{if } x \; op_r \; 0 \text{ goto } n & \end{array} \qquad \begin{array}{rcl} op & ::= & + \mid - \mid * \mid / \\ op_r & ::= & < \mid = \\ P & \in & \mathbb{N} \to I \end{array}$$

Formally defining a translation from a source language such as WHILE to a lower-level intermediate language such as WHILE3ADDR is possible, but more appropriate for the scope of a compilers course. For our purposes, the above should suffice as intuition. We will formally define the semantics of WHILE3ADDR in subsequent lectures.

[2]The idea of the mapping between numbers and instructions is akin to mapping line numbers to code. Other textbooks, such as Nielsen et al.'s Principles of Program Analysis, similarly use abstract *labels* to denote a program point.

2.3 Extensions

The languages described above are sufficient to introduce the fundamental concepts of program analysis in this course. However, we will eventually examine various extensions to WHILE and WHILE3ADDR, so that we can understand how more complicated constructs in real languages can be analyzed. Some of these extensions to WHILE3ADDR will include:

$$
\begin{array}{rcll}
I & ::= & \ldots & \\
 & | & x := f(y) & \textit{function call} \\
 & | & \text{return } x & \textit{return} \\
 & | & x := y.m(z) & \textit{method call} \\
 & | & x := \&p & \textit{address-of operator} \\
 & | & x := *p & \textit{pointer dereference} \\
 & | & *p := x & \textit{pointer assignment} \\
 & | & x := y.f & \textit{field read} \\
 & | & x.f := y & \textit{field assignment}
\end{array}
$$

We will not give semantics to these extensions now, but it is useful to be aware of them as you will see intermediate code like this in practical analysis frameworks.

2.4 Control flow graphs

Many program analysis tools and techniques work on a representation of code known as a *control-flow graph* (CFG), which is a graph-based representation of the flow of control through the program. It connects simple instructions in a way that statically captures all possible execution paths through the program and defines the execution order of instructions in the program. When control could flow in more than one direction, depending on program values, the graph branches. An example is the representation of an `if` or `while` statement. At the end of the instructions in each branch of an `if` statement, the branches merge together to point to the single instruction that comes afterward. Historically, this arises from the use of program analysis to optimize programs.

More precisely, a control flow graph consists of a set of nodes and edges. The nodes $\mathcal{N}$ correspond to *basic blocks*: Sequences of program instructions with no jumps in or out (no gotos, no labeled targets). The edges $\mathcal{E}$ represent the flow of control between basic blocks. We use *Pred(n)* to denote the set of all predecessors of the node *n*, and *Succ(n)* the set of all successors. A CFG has a start node, and a set of final nodes, corresponding to return or other termination of a function. Finally, for the purposes of dataflow analysis, we say that a *program point* exists before and after each node. Note that there exists considerable flexibility in these definitions, and the precision of the representation can vary based on the desired precision of the resulting analysis as well as the peculiarities of the language. In this course, we will in fact often ignore the concept of a basic block and just treat instructions as the nodes in a graph; this view is semantically equivalent and simpler, but less efficient in practice. Further defining and learning how to construct CFGs is a subject best left to a compilers course; this discussion should suffice for our purposes.

Chapter 3

Program Semantics

3.1 Operational Semantics

To reason about analysis correctness, we need a clear definition of what a program *means*. One way to do this is using natural language (e.g., the Java Language Specification). However, although natural language specifications are accessible, they are also often imprecise. This can lead to many problems, including incorrect compiler implementations or program analyses.

A better alternative is a formal definition of program semantics. We begin with *operational semantics*, which mimics, at a high level, the operation of a computer executing the program. Such a semantics also reflects the way that techniques such as dataflow analysis or Hoare Logic reason about the program, so it is convenient for our purposes.

There are two broad classes of operational semantics: *big-step operational semantics*, which specifies the entire operation of a given expression or statement; and *small-step operational semantics*, which specifies the operation of the program one step at a time.

3.1.1 WHILE: Big-step operational semantics

We'll start by restricting our attention to arithmetic expressions, for simplicity. What is the meaning of a WHILE expression? Some expressions, like a natural number, have a very clear meaning: The "meaning" of 5 is just, well, 5. But what about $x + 5$? The meaning of this expression clearly depends on the value of the variable x. We must *abstract* the value of variables as a function from variable names to integer values:

$$E \in \mathit{Var} \to \mathbb{Z}$$

Here E denotes a particular program *state*. The meaning of an expression with a variable like $x + 5$ involves "looking up" the x's value in the associated E, and substituting it in. Given a state, we can write a *judgment* as follows:

$$\langle E, a \rangle \Downarrow n$$

This means that given program state E, the arithmetic expression a evaluates to n. This formulation is called *big-step* operational semantics; the $\Downarrow$ judgment relates an expression and its "meaning."[1] We then build up the meaning of more complex expressions using *rules of inference* (also called *derivation* or *evaluation* rules). An inference rule is made up of a set of judgments above the line, known as *premises*, and a judgment below the line, known as the *conclusion*. The meaning of an inference rule is that the conclusion holds if all of the premises hold:

[1] Note that I have chosen $\Downarrow$ because it is a common notational convention; it's not otherwise special. This is true for many notational choices in formal specification.

$$\frac{premise_1 \quad premise_2 \quad \ldots \quad premise_n}{conclusion}$$

An inference rule with no premises is an *axiom*, which is always true. For example, integers always evaluate to themselves, and the meaning of a variable is its stored value in the state:

$$\frac{}{\langle E, n\rangle \Downarrow n}\ \textit{big-int} \qquad \frac{}{\langle E, x\rangle \Downarrow E(x)}\ \textit{big-var}$$

Addition expressions illustrate a rule with premises:

$$\frac{\langle E, a_1\rangle \Downarrow n_1 \quad \langle E, a_2\rangle \Downarrow n_2}{\langle E, a_1 + a_2\rangle \Downarrow n_1 + n_2}\ \textit{big-add}$$

But, how does the value of x come to be "stored" in E? For that, we must consider WHILE *statements*. Unlike expressions, statements have no direct result. However, they can have *side effects*. That is to say: the "result" or *meaning* of a Statement is a *new state*. The judgment $\Downarrow$ as applied to statements and states therefore looks like:

$$\langle E, S\rangle \Downarrow E'$$

This allows us to write inference rules for statements, bearing in mind that their *meaning* is not an integer, but a new state. The meaning of `skip`, for example, is an unchanged state:

$$\frac{}{\langle E, \texttt{skip}\rangle \Downarrow E}\ \textit{big-skip}$$

Statement sequencing, on the other hand, does involve premises:

$$\frac{\langle E, S_1\rangle \Downarrow E' \quad \langle E', S_2\rangle \Downarrow E''}{\langle E, S_1; S_2\rangle \Downarrow E''}\ \textit{big-seq}$$

The `if` statement involves two rules, one for if the boolean predicate evaluates to `true` (rules for boolean expressions not shown), and one for if it evaluates to `false`. I'll show you just the first one for demonstration:

$$\frac{\langle E, b\rangle \Downarrow \texttt{true} \quad \langle E, S_1\rangle \Downarrow E'}{\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, E\rangle \Downarrow E'}\ \textit{big-iftrue}$$

What should the second rule for `if` *look like?*

This brings us to assignments, which produce a new state in which the variable being assigned to is mapped to the value from the right-hand side. We write this with the notation $E[x \mapsto n]$, which can be read "a new state that is the same as E except that x is mapped to n."

$$\frac{\langle E, a\rangle \Downarrow n}{\langle E, x := a\rangle \Downarrow E[x \mapsto n]}\ \textit{big-assign}$$

Note that the update to the state is modeled *functionally*; the variable E still refers to the old state, while $E[x \mapsto n]$ is the new state represented as a mathematical map.

Fully specifying the semantics of a language requires a judgment rule like this for every language construct. For brevity, these notes only include a subset of the rules for the complete WHILE language.

Exercise 1. What are the rule(s) for the `while` construct?

3.1.2 WHILE: Small-step operational semantics

Big-step operational semantics has its uses. Among other nice features, it directly suggests a simple interpreter implementation for a given language. However, it is difficult to talk about a statement or program whose evaluation does not terminate. Nor does it give us any way to talk about intermediate states (so modeling multiple threads of control is out).

Sometimes it is instead useful to define a *small-step operational semantics*, which specifies program execution one step at a time. We refer to the pair of a statement and a state ($\langle E, S\rangle$) as a *configuration*. Whereas big step semantics specifies program meaning as a function between a configuration and a new state, small step models it as a step from one configuration to another.

You can think of small-step semantics as a set of rules that we repeatedly apply to configurations until we reach a *final configuration* for the language ($\langle E, \texttt{skip}\rangle$, in this case) if ever.[2] We write this new judgment using a slightly different arrow: $\rightarrow$. $\langle E, S\rangle \rightarrow \langle E', S'\rangle$ indicates one step of execution; $\langle E, S\rangle \rightarrow^* \langle E', S'\rangle$ indicates zero or more steps of execution. We formally define multiple execution steps as follows:

$$\frac{}{\langle E, S\rangle \rightarrow^* \langle E, S\rangle}\ \textit{multi-reflexive} \qquad \frac{\langle E, S\rangle \rightarrow \langle E', S'\rangle \quad \langle E', S'\rangle \rightarrow^* \langle E'', S''\rangle}{\langle E, S\rangle \rightarrow^* \langle E'', S''\rangle}\ \textit{multi-inductive}$$

To be complete, we should also define auxiliary small-step operators $\rightarrow_a$ and $\rightarrow_b$ for arithmetic and boolean expressions, respectively; only the operator for statements results in an updated state (as in big step). The types of these judgments are thus:

$$\begin{array}{lcl} \rightarrow & : & (E \times \texttt{Stmt}) \rightarrow (E \times \texttt{Stmt}) \\ \rightarrow_a & : & (E \times \texttt{Aexp}) \rightarrow \texttt{Aexp} \\ \rightarrow_b & : & (E \times \texttt{Bexp}) \rightarrow \texttt{Bexp} \end{array}$$

We can now again write the semantics of a WHILE program as new rules of inference. Some rules look very similar to the big-step rules, just with a different arrow. For example, consider variables:

$$\frac{}{\langle E, x\rangle \rightarrow_a E(x)}\ \textit{small-var}$$

Things get more interesting when we return to statements. Remember, small-step semantics express a single execution step. So, consider an `if` statement:

$$\frac{\langle E, b\rangle \rightarrow_b b'}{\langle E, \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2\rangle \rightarrow \langle E, \texttt{if}\ b'\ \texttt{then}\ S_1\ \texttt{else}\ S_2\rangle}\ \textit{small-if-congruence}$$

$$\frac{}{\langle E, \texttt{if true then}\ S_1\ \texttt{else}\ S_2\rangle \rightarrow \langle E, S_1\rangle}\ \textit{small-iftrue}$$

Exercise 2. We have again omitted the *small-iffalse* case, as well as rule(s) for `while`, as exercises to the reader.

Note also the change for statement sequencing:

$$\frac{\langle E, S_1\rangle \rightarrow \langle E', S_1'\rangle}{\langle E, S_1; S_2\rangle \rightarrow \langle E', S_1'; S_2\rangle}\ \textit{small-seq-congruence}$$

$$\frac{}{\langle E, \texttt{skip}; S_2\rangle \rightarrow \langle E, S_2\rangle}\ \textit{small-seq}$$

[2] Not all statements reach a final configuration, like `while true do skip`.

3.1.3 WHILE3ADDR: Small-step semantics

The ideas behind big- and small-step operational semantics are consistent across languages, but the way they are written can vary based on what is notationally convenient for a particular language or analysis. WHILE3ADDR is slightly different from WHILE, so beyond requiring different rules for its different constructs, it makes sense to modify our small-step notation a bit for defining the meaning of a WHILE3ADDR program.

First, let's revisit the *configuration* to account for the slightly different *meaning* of a WHILE3ADDR program. As before, the configuration must include the state, which we still call E, mapping variables to values. However, a well-formed, terminating WHILE program was effectively a single statement that can be iteratively reduced to `skip`; a WHILE3ADDR program, on the other hand, is a mapping from natural numbers to program instructions. So, instead of a statement that is being reduced in steps, the WHILE3ADDR c must include a program counter n, representing the next instruction to be executed.

Thus, a configuration c of the abstract machine for WHILE3ADDR must include the stored program P (which we will generally treat implicitly), the state environment E, and the current program counter n representing the next instruction to be executed ($c \in E \times \mathbb{N}$). The abstract machine executes one step at a time, executing the instruction that the program counter points to, and updating the program counter and environment according to the semantics of that instruction.

This adds a tiny bit of complexity to the inference rules, because they must explicitly consider the mapping between line number/labels and program instructions. We represent execution of the abstract machine via a judgment of the form $P \vdash \langle E, n\rangle \rightsquigarrow \langle E', n'\rangle$ The judgment reads: "When executing the program P, executing instruction n in the state E steps to a new state E' and program counter n'."[3] To see this in action, consider a simple inference rule defining the semantics of the constant assignment instruction:

$$\frac{P(n) = x := m}{P \vdash \langle E, n\rangle \rightsquigarrow \langle E[x \mapsto m], n+1\rangle}\ \textit{step-const}$$

This states that in the case where the nth instruction of the program P (looked up using $P(n)$) is a constant assignment $x := m$, the abstract machine takes a step to a state in which the state E is updated to map x to the constant m, written as $E[x \mapsto m]$, and the program counter now points to the instruction at the following address $n+1$. We similarly define the remaining rules:

$$\frac{P[n] = x := y}{P \vdash \langle E, n\rangle \rightsquigarrow \langle E[x \mapsto E(y)], n+1\rangle}\ \textit{step-copy}$$

$$\frac{P(n) = x := y\ op\ z \quad E(y)\ \mathbf{op}\ E(z) = m}{P \vdash \langle E, n\rangle \rightsquigarrow \langle E[x \mapsto m], n+1\rangle}\ \textit{step-arith}$$

$$\frac{P(n) = \text{goto } m}{P \vdash \langle E, n\rangle \rightsquigarrow \langle E, m\rangle}\ \textit{step-goto}$$

$$\frac{P(n) = \text{if } x\ op_r\ 0 \text{ goto } m \quad E(x)\ \mathbf{op_r}\ 0 = true}{P \vdash \langle E, n\rangle \rightsquigarrow \langle E, m\rangle}\ \textit{step-iftrue}$$

$$\frac{P(n) = \text{if } x\ op_r\ 0 \text{ goto } m \quad E(x)\ \mathbf{op_r}\ 0 = false}{P \vdash \langle E, n\rangle \rightsquigarrow \langle E, n+1\rangle}\ \textit{step-iffalse}$$

[3] I could have used the same $\rightarrow$ I did above instead of $\rightsquigarrow$, but I don't want you to mix them up.

3.1.4 Derivations and provability

Among other things, we can use operational semantics to prove that concrete program expressions will evaluate to particular values. We do this by chaining together rules of inference (which simply list the hypotheses necessary to arrive at a conclusion) into *derivations*, which interlock instances of rules of inference to reach particular conclusions. For example:

$$\dfrac{\dfrac{\langle E_1, 4\rangle \Downarrow 4 \quad \langle E_1, 2\rangle \Downarrow 2}{\langle E_1, 4 * 2\rangle \Downarrow 8} \qquad \langle E_1, 6\rangle \Downarrow 6}{\langle E_1, (4 * 2) - 6\rangle \Downarrow 2}$$

We say that $\langle E, a\rangle \Downarrow n$ is *provable* (expressed mathematically as $\vdash \langle E, a\rangle \Downarrow n$) if there exists a well-formed derivation with $\langle E, a\rangle \Downarrow n$ as its conclusion. "Well formed" simply means that every step in the derivation is a valid instance of one of the rules of inference for this system.

A proof system like our operational semantics is *complete* if every true statement is provable. It is *sound* (or *consistent*) if every provable judgment is true.

3.2 Proof techniques using operational semantics

A precise language specification lets us precisely prove properties of our language or programs written in it (and analyses of those programs!). Note that this exposition primarily uses big-step semantics to illustrate, but the concepts generalize.

Well-founded induction. A key family of proof techniques in programming languages is based on *induction*. You may already be familiar with *mathematical induction*. As a reminder: if $P(n)$ is a property of the natural numbers that we want to show holds for all n, mathematical induction says that it suffices to show that $P(0)$ is true (the base case), and then that if $P(m)$ is true, then so is $P(m + 1)$ for any natural number m (the inductive step). This works because there are no infinite descending chains of natural numbers. So, for any n, $P(n)$ can be obtained by simply starting from the base case and applying n instances of the inductive step.

Mathematical induction is a special case of *well-founded induction*, a general, powerful proof principle that works as follows: a relation $\prec \subseteq A \times A$ is well-founded if there are no infinite descending chains in A. If so, to prove $\forall x \in A.P(x)$ it is enough to prove $\forall x \in A.[\forall y \prec x \Rightarrow P(y)] \Rightarrow P(x)$; the base case arises when there is no $y \prec x$, and so the part of the formula within the brackets [] is vacuously true.[4]

Structural induction. *Structural induction* is another special case of well-founded induction where the $\prec$ relation is defined on the structure of a program or a derivation. For example, consider the syntax of arithmetic expressions in WHILE, `Aexp`. Induction on a recursive definition like this proves a property about a mathematical structure by demonstrating that the property holds for all possible forms of that structure. We define the relation $a \prec b$ to hold if a is a substructure of b. For `Aexp` expressions, the relation $\prec \subseteq$ `Aexp` $\times$ `Aexp` is:

$$a_1 \prec a_1 + a_2$$
$$a_1 \prec a_1 * a_2$$
$$a_2 \prec a_1 + a_2$$
$$a_2 \prec a_1 * a_2$$
...etc., for all arithmetic operators op_a

To prove that a property P holds for all arithmetic expressions in WHILE (or, $\forall a \in$ `Aexp`$.P(a)$), we must show P holds for both the base cases and the inductive cases. a is a

[4]Mathematical induction as a special case arises when $\prec$ is simply the predecessor relation ($(x, x + 1)|x \in \mathbb{N}$).

base case if there is no a' such that $a' \prec a$; a is an inductive case if $\exists a' \,.\, a' \prec a$. There is thus one proof case per form of the expression. For `Aexp`, the base cases are:

$$\vdash \forall n \in Z \,.\, P(n)$$
$$\vdash \forall x \in \text{Vars} \,.\, P(x)$$

And the inductive cases:

$$\vdash \forall a_1, a_2 \in \texttt{Aexp} \,.\, P(a_1) \wedge P(a_2) \Rightarrow P(a_1 + a_2)$$
$$\vdash \forall a_1, a_2 \in \texttt{Aexp} \,.\, P(a_1) \wedge P(a_2) \Rightarrow P(a_1 * a_2)$$
. . . and so on for the other arithmetic operators. . .

Example. Let $L(a)$ be the number of literals and variable occurrences in some expression a and $O(a)$ be the number of operators in a. Prove by induction on the structure of a that $\forall a \in \texttt{Aexp} \,.\, L(a) = O(a) + 1$:

Base cases:

- Case $a = n$. $L(a) = 1$ and $O(a) = 0$
- Case $a = x$. $L(a) = 1$ and $O(a) = 0$

Inductive case 1: Case $a = a_1 + a_2$

- By definition, $L(a) = L(a_1) + L(a_2)$ and $O(a) = O(a_1) + O(a_2) + 1$.
- By the induction hypothesis, $L(a_1) = O(a_1) + 1$ and $L(a_2) = O(a_2) + 1$.
- Thus, $L(a) = O(a_1) + O(a_2) + 2 = O(a) + 1$.

The other arithmetic operators follow the same logic.

Other proofs for the expression sublanguages of WHILE can be similarly conducted. For example, we could prove that the small-step and big-step semantics will obtain equivalent results on expressions:

$$\forall a \in \texttt{AExp} \,.\, \langle E, a\rangle \rightarrow_a^* n \Leftrightarrow \langle E, a\rangle \Downarrow n$$

The actual proof is left as an exercise, but note that this works because the semantics rules for expressions are strictly syntax-directed: the meaning of an expression is determined entirely by the meaning of its subexpressions, the structure of which guides the induction.

Induction on the structure of derivations. Unfortunately, that last statement is *not* true for *statements* in the WHILE language. For example, imagine we'd like to prove that WHILE is *deterministic* (that is, if a statement terminates, it always evaluates to the same value). More formally, we want to prove that:

$$\forall a \in \texttt{Aexp} \,.\ \forall E \,.\, \forall n, n' \in \mathbb{N} \,.\quad \langle E, a\rangle \Downarrow n \wedge \langle E, a\rangle \Downarrow n' \Rightarrow n = n' \tag{3.1}$$
$$\forall b \in \texttt{Bexp} \,.\ \forall E \,.\, \forall b, b' \in \mathcal{B} \,.\quad \langle E, b\rangle \Downarrow b \wedge \langle E, b\rangle \Downarrow b' \Rightarrow b = b' \tag{3.2}$$
$$\forall S \,.\qquad \forall E, E', E'' \,.\qquad \langle E, S\rangle \Downarrow E' \wedge \langle E, S\rangle \Downarrow E'' \Rightarrow E' = E'' \tag{3.3}$$

We can't prove the third statement with structural induction on the language syntax because the evaluation of statements (like `while`) does *not* depend only on the evaluation of its subexpressions.

Fortunately, there is another way. Recall that the operational semantics assign meaning to programs by providing rules of inference that allow us to prove judgments by making derivations. Derivation trees (like the expression trees we discussed above) are also defined inductively, and are built of sub-derivations. Because they have structure, we can again use structural induction, but here, on the structure of derivations.

Instead of assuming (and reasoning about) some statement S, we instead assume a derivation $D :: \langle E, S\rangle \Downarrow E'$ and induct on the structure of that derivation (we define $D ::$ judgment to mean "D is the derivation that proves judgment." e.g., $D :: \langle E, x + 1\rangle \Downarrow 2$). That is, to prove that property P holds for a statement, we will prove that P holds for all possible derivations of that statement. Such a proof consists of the following steps:
Base cases: show that P holds for each atomic derivation rule with no premises (of the form $\overline{J}$).
Inductive cases: For each derivation rule of the form

$$\frac{H_1 ... H_n}{J}$$

By the induction hypothesis, P holds for H_i, where $i = 1 \ldots n$. We then have to prove that the property is preserved by the derivation using the given rule of inference.

A key technique for induction on derivations is *inversion.* Because the number of forms of rules of inference is finite, we can tell which inference rules might have been used last in the derivation. For example, given $D :: \langle E_i, x := 55\rangle \Downarrow E$, we know (by inversion) that the assignment rule of inference must be the last rule used in D (because no other rules of inference involve an assignment statement in their concluding judgment). Similarly, if $D :: \langle E_i, \text{while } b \text{ do } S\rangle \Downarrow E$, then (by inversion) the last rule used in D was either the `while-true` rule or the `while-false` rule.

Given those preliminaries, to prove that the evaluation of statements is deterministic (equation (3) above), pick arbitrary S, E, E', and $D :: \langle E, S\rangle \Downarrow E'$

Proof: by induction of the structure of the derivation D, which we define $D :: \langle E, S\rangle \Downarrow E'$.

Base case: the one rule with no premises, `skip`:

$$D :: \overline{\langle E, \texttt{skip}\rangle \Downarrow E}$$

By inversion, the last rule used in D' (which, again, produced E'') must also have been the rule for `skip`. By the structure of the `skip` rule, we know $E'' = E$.

Inductive cases: We need to show that the property holds when the last rule used in D was each of the possible non-skip WHILE commands. I will show you one representative case; the rest are left as an exercise. If the last rule used was the `while-true` statement:

$$D :: \frac{D_1 :: \langle E, b\rangle \Downarrow \texttt{true} \quad D_2 :: \langle E, S\rangle \Downarrow E_1 \quad D_3 :: \langle E_1, \texttt{while } b \texttt{ do } S\rangle \Downarrow E'}{\langle E, \texttt{while } b \texttt{ do } S\rangle \Downarrow E'}$$

Pick arbitrary E'' such that $D'' :: \langle E, \texttt{while } b \texttt{ do } S\rangle \Downarrow E''$

By inversion, and determinism of boolean expressions, D'' must also use the same `while-true` rule. So D'' must also have subderivations $D_2'' :: \langle E, S\rangle \Downarrow E_1''$ and $D_3'' :: \langle E_1'', \texttt{while } b \texttt{ do } S\rangle \Downarrow E''$. By the induction hypothesis on D_2 with D_2'', we know $E_1 = E_1''$. Using this result and the induction hypothesis on D_3 with D_3'', we have $E'' = E'$.

Chapter 4

A Dataflow Analysis Framework for WHILE3ADDR

4.1 Defining a dataflow analysis

A dataflow analysis computes some dataflow information at each program point in the control flow graph.[1] We thus start by examining how this information is defined. We will use σ to denote this information. Typically σ tells us something about each variable in the program. For example, σ may map variables to abstract values taken from some set L:

$$\sigma \in \mathit{Var} \to L$$

L represents the set of abstract values we are interested in tracking in the analysis. This varies from one analysis to another. For example, consider a *zero analysis*, which tracks whether each variable is zero or not at each program point (Thought Question: Why would this be useful?). For this analysis, we define L to be the set $\{Z, N, \top\}$. The abstract value Z represents the value 0, N represents all nonzero values. $\top$ is pronounced "top", and we define it more concretely later it in these notes; we use it as a question mark, for the situations when we do not know whether a variable is zero or not, due to imprecision in the analysis.

Conceptually, each abstract value represents a set of one or more concrete values that may occur when a program executes. We define an abstraction function α that maps each possible concrete value of interest to an abstract value:

$$\alpha : \mathbb{Z} \to L$$

For zero analysis, we define α so that 0 maps to Z and all other integers map to N:

$$\alpha_Z(0) = Z$$
$$\alpha_Z(n) = N \text{where } n \neq 0$$

The core of any program analysis is how individual instructions in the program are analyzed and affect the analysis state σ at each program point. We define this using *flow functions* that map the dataflow information at the program point immediately *before* an instruction to the dataflow information *after* that instruction. A flow function should represent the semantics of the instruction, but abstractly, in terms of the abstract values tracked by the analysis. We will link semantics to the flow function precisely when we talk about correctness of dataflow analysis. For now, to approach the idea by example, we define the flow functions f_Z for zero analysis on WHILE3ADDR as follows:

[1] Refer to the first set of course notes for an overview of CFGs.

$$f_Z[\![x := 0]\!](\sigma) \quad = \sigma[x \mapsto Z] \tag{4.1}$$
$$f_Z[\![x := n]\!](\sigma) \quad = \sigma[x \mapsto N] \text{ where } n \neq 0 \tag{4.2}$$
$$f_Z[\![x := y]\!](\sigma) \quad = \sigma[x \mapsto \sigma(y)] \tag{4.3}$$
$$f_Z[\![x := y \; op \; z]\!](\sigma) \quad = \sigma[x \mapsto \top] \tag{4.4}$$
$$f_Z[\![\text{goto } n]\!](\sigma) \quad = \sigma \tag{4.5}$$
$$f_Z[\![\text{if } x = 0 \text{ goto } n]\!](\sigma) \quad = \sigma \tag{4.6}$$

In the notation, the form of the instruction is an implicit argument to the function, which is followed by the explicit dataflow information argument, in the form $f_Z[\![I]\!](\sigma)$. (1) and (2) are for assignment to a constant. If we assign 0 to a variable x, then we should update the input dataflow information σ so that x maps to the abstract value Z. The notation $[x \mapsto Z]\sigma$ denotes dataflow information that is identical to σ except that the value in the mapping for x is updated to refer to Z. Flow function (3) is for copies from a variable y to another variable x: we look up y in σ, written $\sigma(y)$, and update σ so that x maps to the same abstract value as y.

We start with a generic flow function for arithmetic instructions (4). Arithmetic can produce either a zero or a nonzero value, so we use the abstract value $\top$ to represent our uncertainty. More precise flow functions are available based on certain instructions or operands. For example, if the instruction is subtraction and the operands are the same, the result will definitely be zero. Or, if the instruction is addition, and the analysis information tells us that one operand is zero, then the addition is really a copy and we can use a flow function similar to the copy instruction above. These examples could be written as follows (we would still need the generic case above for instructions that do not fit such special cases):

$$\begin{aligned} f_Z[\![x := y - y]\!](\sigma) &= \sigma[x \mapsto Z] \\ f_Z[\![x := y + z]\!](\sigma) &= \sigma[x \mapsto \sigma(y)] \quad \text{where } \sigma(z) = Z \end{aligned}$$

Exercise 1. Define another flow function for some arithmetic instruction and certain conditions where you can also provide a more precise result than $\top$.

The flow function for branches ((5) and (6)) is trivial: branches do not change the state of the machine other than to change the program counter, and thus the analysis result is unaffected.

However, we can provide a better flow function for conditional branches if we distinguish the analysis information produced when the branch is taken or not taken. To do this, we extend our notation once more in defining flow functions for branches, using a subscript to the instruction to indicate whether we are specifying the dataflow information for the case where the condition is true (T) or when it is false (F). For example, to define the flow function for the true condition when testing a variable for equality with zero, we use the notation $f_Z[\![\text{if } x = 0 \text{ goto } n]\!]_T(\sigma)$. In this case we know that x is zero so we can update σ with the Z lattice value. Conversely, in the false condition we know that x is nonzero:

$$\begin{aligned} f_Z[\![\text{if } x = 0 \text{ goto } n]\!]_T(\sigma) &= \sigma[x \mapsto Z] \\ f_Z[\![\text{if } x = 0 \text{ goto } n]\!]_F(\sigma) &= \sigma[x \mapsto N] \end{aligned}$$

Exercise 2. Define a flow function for a conditional branch testing whether a variable $x < 0$.

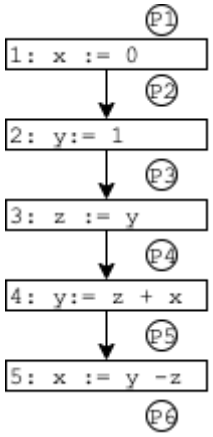
4.2 Running a dataflow analysis

The point of developing a dataflow analysis is to compute information about possible program states at each point in a program. For example, for of zero analysis, whenever we divide some expression by a variable x, we might like to know whether x must be zero (the abstract value Z) or may be zero (represented by $\top$) so that we can warn the developer.

4.2.1 Straightline code

One way to think of a simple dataflow analysis is that are statically simulating program execution, tracking only the information we care about. For each node in the CFG (each of which contains an instruction), we use the flow function to compute the dataflow analysis information at the program point immediately *after* that node from the information we had at the program point *before* that node. To demonstrate, consider the following simple program (left), with its control flow graph (middle):

$1: \quad x := 0$
$2: \quad y := 1$
$3: \quad z := y$
$4: \quad y := z + x$
$5: \quad x := y - z$



	x	y	z
P1	?	?	?
P2	Z	?	?
P3	Z	N	?
P4	Z	N	N
P5	Z	N	N
P6	$\top$	N	N

For such simple code, we can track analysis information using a table with a column for each program variable and a row for each program point (right, above).

The first thing to notice is that, because flow functions operate on the abstract state for the program point immediately before a node, we need some kind of *initial* assumption (this confusion is illustrated by the ? in the cells of the table). We will return to this point in a moment, since those values don't influence the analysis for such simple, straight-line code.

Notice also that the analysis is imprecise at the end with respect to the value of x. We were able to keep track of which values are zero and nonzero quite well through instruction 4, using (in the last case) the flow function that knows that adding a variable known to be zero is equivalent to a copy. However, at instruction 5, the analysis does not know that y and z are equal, and so it cannot determine whether x will be zero. Because the analysis is not tracking the exact values of variables, but rather approximations, it will inevitably be imprecise in certain situations. However, in practice, well-designed approximations can often allow dataflow analysis to compute quite useful information.

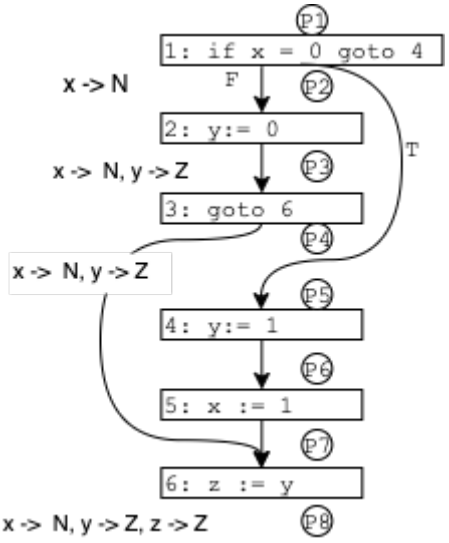
4.2.2 Alternative Paths: Illustration

Things get more interesting in WHILE3ADDR code that contains `if` statements. An `if` statement introduces two possible paths through the program. Consider the following simple example (left), and its CFG (middle).[2] We will begin by analyzing the first node as though the

[2] A point on diagrams: in the interest of clarity, we sometimes elide program points between nodes when we can. That is, in this example,, the state going into instruction 3 is exactly the state coming out of instruction 2, so we label a single program point P3. However, when we need to consider multiple paths to determine the incoming state at a node, we often need differentiate the two program points in our CFG diagrams.

branch is not taken:

$1:$ if $x = 0$ goto 4
$2:$ $y := 0$
$3:$ goto 6
$4:$ $y := 1$
$5:$ $x := 1$
$6:$ $z := y$

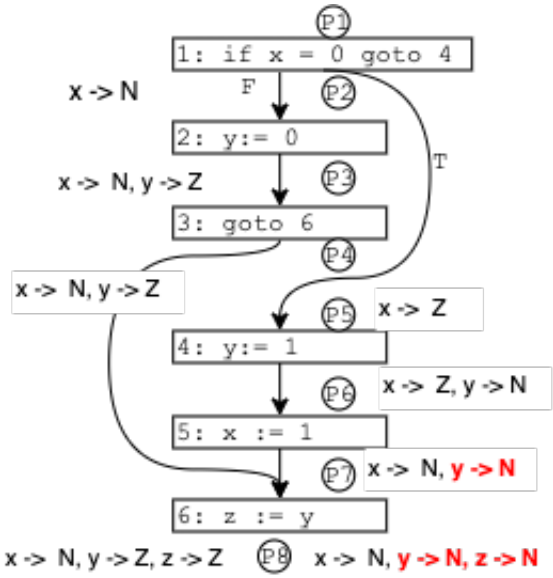


	x	y	z
P1	?	?	?
P2	N_F	?	?
P3	N	Z	?
P4	N	Z	?
P5			
P6			
P7	N?	Z?	??
P8	N??	Z??	Z??

In the table above, the entry for x at P2 indicates the abstract value produced for the false condition on the branch, which is then used as input to analyze instruction 2 (and produce the state at P3). We can go right from P3 to P4 without any complexity. But, if we just continue "simulating" execution, we get to P7. It has *two* possible incoming edges, so two possible incoming states to use for the flow function for instruction 6. What to do? We have not yet analyzed a path through lines 4 and 5. The table shows the (questionable) values if we just use the state coming from P4 as "incoming" at instruction 6, and ignore what might have happened along that other path.

Perhaps turning to that alternative path, will give answers. Let's analyze instructions 4 and 5 as if we had taken the true branch at instruction 1:

	x	y	z	
P1	?	?	?	
P2	Z_T, N_F	?	?	
P3	N	Z	?	
P4	N	Z	?	
P5	Z	?	?	
P6	Z	N	?	
P7	N	N?	?	*note: different!*
P8	N??	N??	N??	*??????*

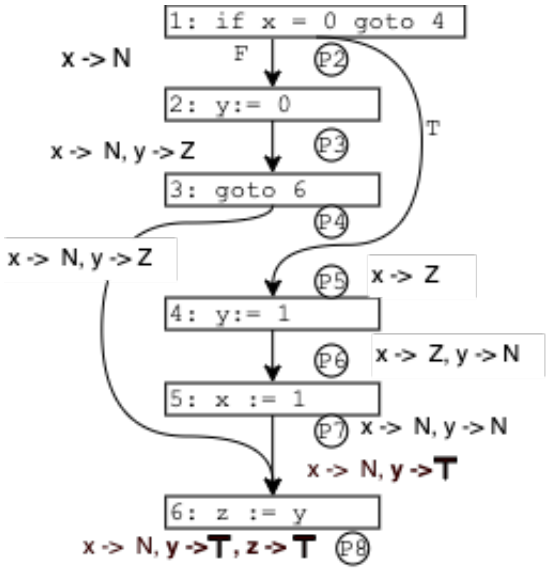


We have a dilemma. The first time we analyzed instruction 6, the incoming state had come from instruction 3, where x was nonzero and y was zero. Now have, the incoming state coming from instruction 5 is different: x is still nonzero, but so is y!

We resolve this dilemma by *combining* the abstract values computed along the two paths for y. The incoming abstract values at P7 for y are N and Z. We represent this uncertainty with a new abstract value $\top$ (pronounced "top"). This value indicates that we do know know if y is zero or not, because we don't know how we reached this program location. We can apply similar logic to x, but because x is nonzero on both incoming paths, we can maintain our knowledge that x is nonzero. Thus, we should analyze instruction 6 with this combined incoming state: $\{x \mapsto N, y \mapsto \top\}$.

The corrected analysis, showing the *combined* state at P6, looks like:

	x	y	z	
P1	?	?	?	
P2	Z_T, N_F	?	?	
P3	N	Z	?	
P4	N	Z	?	
P5	Z	?	?	
P6	Z	N	?	
P7	N	$\top$	?	*combined with P4*
P8	N	$\top$	$\top$	*corrected*



4.3 Join

The mechanism for combining analysis results along multiple paths is called a *join* operation, $\sqcup$. When taking two abstract values $l_1, l_2 \in L$, the result of $l_1 \sqcup l_2$ is an abstract value l_j that

generalizes both l_1 and l_2.

To precisely define what "generalizes" means, we define a partial order $\sqsubseteq$ over abstract values, and say that l_1 and l_2 are at least as precise as l_j, written $l_1 \sqsubseteq l_j$. Recall that a partial order is any relation that is:

- reflexive: $\forall l : l \sqsubseteq l$
- transitive: $\forall l_1, l_2, l_3 : l_1 \sqsubseteq l_2 \wedge l_2 \sqsubseteq l_3 \Rightarrow l_1 \sqsubseteq l_3$
- anti-symmetric: $\forall l_1, l_2 : l_1 \sqsubseteq l_2 \wedge l_2 \sqsubseteq l_1 \Rightarrow l_1 = l_2$

A set of values L that is equipped with a partial order $\sqsubseteq$, and for which the least upper bound of any two values in that ordering $l_1 \sqcup l_2$ is unique and is also in L, is called a *join-semilattice*. We require that the abstract values used in dataflow analyses form a join-semilattice. We will use the term lattice for short; as we will see below, this is the correct terminology for most dataflow analyses anyway.

For zero analysis, we define the partial order with $Z \sqsubseteq \top$ and $N \sqsubseteq \top$, where $Z \sqcup N = \top$. We usually use the symbol $\top$ (pronounced "top") to refer the *maximal* element of a lattice; that is, for all l, we have $l \sqsubseteq \top$. Intuitively, $\top$ is the most general dataflow value.

We have now considered all the elements necessary to define a dataflow analysis:

- a lattice $(L, \sqsubseteq)$
- an abstraction function α
- a flow function f
- initial dataflow analysis assumptions, σ_0

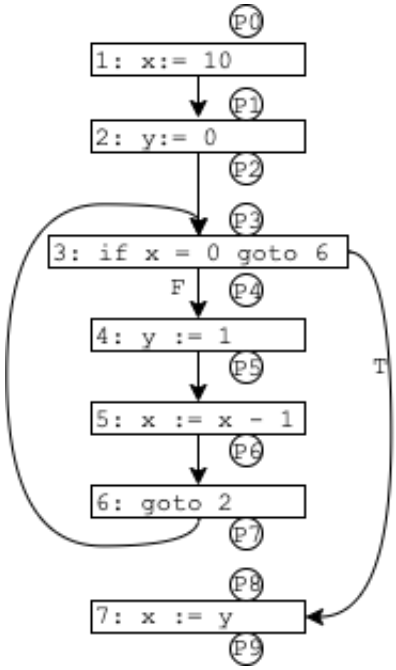
Note that the theory of lattices answers that side question that came up in the very first example: what should we assume about the value of input variables (the question marks in our example tables)? If we do not know anything about the value x can be, one good choice is to assume it can be anything. That is, in the initial environment σ_0, variables' initial state is mapped to $\top$.

4.3.1 Dataflow analysis of loops

We now consider WHILE3ADDR programs with loops. Our intuition above, which simply analyzed the two paths induced by the `if` statement separately, no longer works so well. A loop produces a potentially unbounded number of program paths, and we want our analysis to take only bounded time. Consider the following simple looping example:[3]

[3] I provide the CFG for reference but omit the annotations in the interest of a cleaner diagram. Notice that I differentiate P2 and P3 because of the join, as well as P7 and P8, since they don't both come from instruction 6.

1 : $x := 10$
2 : $y := 0$
3 : if $x = 0$ goto 7
4 : $y := 1$
5 : $x := x - 1$
6 : goto 3
7 : $x := y$



	x	y	
P0	$\top$	$\top$	
P1	N	$\top$	
P2	N	Z	
P3	N	Z	*first time through...*
P4	N_F	Z	
P5	N	N	
P6	$\top$	N	
P7	$\top$	N	
P8	Z_t	N	*first time through...*
P9	N	N	*first time through...*

The right-hand side above shows the straightforward straight-line analysis of the path that runs the loop exactly once. Thinking back to our handling of `if` above, we might now reconsider instruction 3, joining the states at P2 and P7 to create a new P3. For x, $N \sqcup \top = \top$. For y, $Z \sqcup N = \top$. This changes the incoming values at instruction 3. We can now choose between two paths once again. We will choose (arbitrarily, for now) to stay within the loop, and reconsider instruction 4. We have new incoming information (at P4, where both x and y are now $\top$). But, since instruction 4 assigns 1 to y, we still know that y is nonzero at P5. The updated input data does not change the analysis results at P5.

A quick check shows that going through the remaining instructions in the loop, even back to instruction 3, the analysis information will no longer change. That is because the flow functions are deterministic: given the same input analysis information and the same instruction, they will produce the same output analysis information.

We say that the dataflow analysis has reached a *fixed point* (or fixpoint). In mathematics, a fixed point of a function is a data value v that is mapped to itself by the function, i.e., $f(v) = v$. In analysis, the mathematical function is the flow function, and the fixed point is a tuple of the dataflow analysis values at each program point. If we invoke the flow function on the fixed point, the analysis results do not change (we get the same fixed point back).

Once we have reached a fixed point for the loop, further analysis of the loop will not be useful. Therefore, we will proceed to analyze statement 7. The final analysis results are as follows:

	x	y	
P0	$\top$	$\top$	
P1	N	$\top$	
P2	N	Z	
P3	$\top$	$\top$	*join*
P4	N_F	$\top$	*updated*
P5	N	N	*already at fixed point*
P6	$\top$	N	*already at fixed point*
P7	$\top$	N	*already at fixed point*
P8	Z_T	$\top$	*updated*
P9	$\top$	$\top$	*updated*

Quickly simulating a run of the program program shows that these results correctly approximate actual execution. The uncertainty in the value of x at P6 and P7 is real: x *is* nonzero after these instructions, except the last time through the loop, when it is zero. The uncertainty in the value of y at the end shows analysis imprecision: this loop always executes at least once, so y will be nonzero at these points. However, the analysis (as currently formulated) cannot tell this for certain, so it reports that it cannot tell if y is zero or not. This is safe—it is always correct to say the analysis is uncertain—but not as precise as would be ideal.

The benefit of analysis, however, is that we can gain correct information about all possible executions of the program with only a finite amount of work. In our example, we only had to analyze the loop statements at most twice each before reaching a fixed point. This is a significant improvement over the actual program execution, which runs the loop 10 times. We sacrificed precision in exchange for coverage of all possible executions, a classic tradeoff.

How can we be confident that the results of the analysis are correct, besides simulating every possible run of a (possibly very complex) program? The intuition behind correctness is the invariant that at each program point, the analysis results approximate all the possible program values that could exist at that point. If the analysis information at the beginning of the program correctly approximates the program arguments, then the invariant is true at the beginning of program execution. One can then make an inductive argument that the invariant is preserved. In particular, when the program executes an instruction, the instruction modifies the program's state. As long as the flow functions account for every possible way that instruction can modify state, then at the analysis fixed point they will have correctly approximated actual program execution. We will make this argument more precise in a future lecture.

4.3.2 A convenience: the $\bot$ abstract value and complete lattices

To define an algorithm for dataflow anlaysis more precisely, we need to be more concrete about how to compute incoming states for CFG nodes with multiple incoming edges (like instruction 3, above). We've been ignoring these in our "one path at a time" approach so far, but this is a handwave for didactic purposes.

Instead, it is more precise and consistent to say that analyzing an instruction *always* uses the incoming dataflow analysis information from *all* instructions that could precede it. However, for instruction 3, this requires a dataflow value from instruction 6, even if instruction 6 has not yet been analyzed. We could do this if we had a dataflow value that is always ignored when it is joined with any other dataflow value. In other words, we need a abstract dataflow value $\bot$ (pronounced "bottom") such that $\bot \sqcup l = l$.

$\bot$ plays a dual role to the value $\top$: it sits at the bottom of the dataflow value lattice. For all l, we have the identity $l \sqsubseteq \top$ and correspondingly $\bot \sqsubseteq l$. There is an greatest lower bound operator *meet*, $\sqcap$, which is dual to $\sqcup$. The meet of all dataflow values is $\bot$.

A set of values L that is equipped with a partial order $\sqsubseteq$, and for which both a least element $\bot$ and a greatest element $\top$ exist in L is called a *complete lattice*.

This provides an elegant solution to the problem mentioned above. We initialize σ at every program point in the program, except at entry, to $\bot$, indicating that the instruction there has not yet been analyzed. We can then *always* merge all input values to a node, whether or not the sources of those inputs have been analysed, because we know that any $\bot$ values from unanalyzed sources will simply be ignored by the join operator $\sqcup$, and that if the dataflow value for that variable will change, we will get to it before the analysis is completed.

4.4 Analysis execution strategy

Our informal strategy above, which considers all paths until the dataflow analysis information reaches a fixed point, can be simplified. The argument for correctness outlined above implies

that for correct flow functions, it doesn't matter how we get to the analysis fixed point (it would be surprising if analysis correctness depended on which branch of an `if` statement we explored first!). It is in fact possible to run the analysis on program instructions in any order we choose. As long as we continue doing so until a reaching a fixed point, the final result will be correct. The simplest correct algorithm for executing dataflow analysis can therefore be stated as follows:

```
for Node n in cfg
    results[n] = ⊥
results[0] = initialDataflowInformation

while not at fixed point
    pick a node n in program
    input = join { results[j] | j in predecessors(n) }
    output = flow(n, input)
    results[n] = output
```

Or, equivalently:

```
for Node n in cfg
    input[n] = ⊥
input[1] = initialDataflowInformation

while not at fixed point
    pick a node n in program
    output = flow(n, input[n])
    for Node j in sucessors(n)
        input[j] = input[j] ⊔ output
```

In the code above, the termination condition is expressed abstractly ("not at fixed point"). It can easily be checked by keeping track, when we process each node, whether the new results have changed compared to what we previously had stored for that node. If the results do not change for any node, the analysis has reached a fix point.

How do we know the algorithm will terminate? The intuition is as follows. We rely on the choice of a node to be fair, so that each node is eventually considered. As long as the analysis is not at a fixed point, some node can be analyzed to produce new results. If our flow functions are well-behaved (technically, if they are monotone, as we will discuss in a future lecture) then each time the flow function runs on a given node, either the results do not change, or they get become more approximate (i.e., they are higher in the lattice). Later runs of the flow function consider more possible paths through the program and therefore produce a more approximate result which considers all these possibilities. If the lattice is of finite height—meaning there are at most a finite number of steps from any place in the lattice going up towards the $\top$ value—then this process must terminate eventually. More concretely: once an abstract value is computed to be $\top$, it will stay $\top$ no matter how many times the analysis is run. The abstraction only flows in one direction.

Although the simple algorithm above always terminates and results in the correct answer, it is still not always the most efficient. Typically, for example, it is beneficial to analyze the program instructions in order, so that results from earlier instructions can be used to update the results of later instructions. It is also useful to keep track of a list of instructions for which there has been a change since the instruction was last analyzed in the result dataflow information of some predecessor. Only those instructions need be analyzed; reanalyzing other instructions is useless since their input has not changed. Kildall captured this intuition with his worklist algorithm, described in pseudocode below. For this algorithm, we associate with each CFG node (or equivalently, each program instruction), two sets of dataflow values representing the

program state just before as well as just after executing the corresponding instruction; these are named `input` and `output` respectively.

```
worklist = ∅
for  Node n in cfg
    input[n] = output[n] = ⊥
    add n to worklist
input[0] = initialDataflowInformation

while worklist is not empty
    take a Node n off the worklist
    output[n] = flow(n, input[n])
    for Node j in succs(n)
          newInput = input[j] ⊔ output[n]
          if newInput ≠ input[j]
                input[j] = newInput
                add j to worklist
```

The algorithm above is very close to the generic algorithm declared previously, except the worklist that chooses the next instruction to analyze and determines when a fixed point is reached. The worklist is initialized with all nodes, so that we visit each of them at least once[4]. The algorithm terminates when there are no more nodes left to process in the worklist.

We can reason about the performance of this algorithm as follows. We only add a node to the worklist when the input data to it changes. The input for a given node can only change h times, where h is the height of the lattice. Thus we add at most $n * h$ nodes to the worklist, where n is the number of nodes/instructions in the program. After running the flow function for a node, however, we must test all its successors to find out if their input has changed. This test is done once for each edge, for each time that the source node of the edge is added to the worklist: thus at most $e * h$ times, where e is the number of control flow edges in the successor graph between instructions. If each operation (such as a flow function, a $\sqcup$ operation, or a comparison of dataflow values) has cost $O(c)$, then the overall cost is $O(c * (n + e) * h)$, or $O(c * e * h)$ because n is bounded by e. Note that c and h are both related to the size of the dataflow lattice; for most analyses, these values increase with the size of the program.

The algorithm above is still abstract: We have not defined the operations to add and remove instructions from the worklist. We would like adding to the work list to be a set addition operation, so that no instruction appears in it multiple times. If we have just analysed the program with respect to an instruction, analyzing it again will not produce different results.

That leaves a choice of which instruction to remove from the worklist. We could choose among several policies, including last-in-first-out (LIFO) order or first-in-first-out (FIFO) order. In practice, the most efficient approach is to identify the strongly-connected components (i.e. loops) in the control flow graph of components and process them in topological order, so that loops that are nested, or appear in program order first, are solved before later loops. This works well because we do not want to do a lot of work bringing a loop late in the program to a fixed point, then have to redo that work when dataflow information from an earlier loop changes.

Within each loop, the instructions should be processed in reverse postorder, the reverse of the order in which each node is last visited when traversing a tree. Consider the example from Section 4.2.2 above, in which instruction 1 is an `if` test, instructions 2–3 are the then branch, instructions 4–5 are the else branch, and instruction 6 comes after the `if` statement. A tree traversal might go as follows: 1, 2, 3, 6, 3 (again), 2 (again), 1 (again), 4, 5, 4 (again), 1 (again). Some instructions in the tree are visited multiple times: once going down, once

[4]If the output of flow functions cannot be $\top$, then we can also initialize the worklist to only the start node.

between visiting the children, and once coming up. The postorder, or order of the last visits to each node, is 6, 3, 2, 5, 4, 1. The reverse postorder is the reverse of this: 1, 4, 5, 2, 3, 6. Now we can see why reverse postorder works well: we explore both branches of the if statement (4–5 and 2–3) before we explore node 6. This ensures that we do not have to reanalyze node 6 after one of its inputs changes.

Although analyzing code using the strongly-connected component and reverse postorder heuristics improves performance substantially in practice, it does not change the worst-case performance results described above.

Chapter 5

Dataflow Analysis Examples

Zero analysis is useful for simply tracking whether a given variable is zero or not; even this simple didactic example can be used to find possible bugs in programs. We will now examine several more complex analyses, including certain well-known analyses that are, particularly (but not exclusively) useful within a compiler.

5.1 Integer Sign Analysis

Integer sign analysis tracks whether each integer in the program is positive, negative, or zero. The results of this analysis can be used to optimize a program or to circumvent errors like using a negative index into an array (or memory underflow gnerally). This analysis is broadly similar to the zero analysis discussed previously (ignoring the possibility of integer overflow, i.e., consider mathematical integers).

This problem admits natural alternatives in designing our analysis, starting with the abstract domain L. For example, we can prefer simplicity in favor of imprecision, defining L to track only whether a value is less than zero, greater than zero, equal to zero, or unknown.

Exercise 1. Specify this on paper, indicating (A) the set of lattice elements, (B) the relation between them, (c) the top element and (d) the bottom element.

One way to increase precision in this analysis is to define a more precise abstract domain. For example, we might decide to track whether a value is less than zero, greater than zero, equal to zero, greater than or equal to zero, less than or equal to zero, non-zero, or unknown. In addition to (trivially) increasing the size of L, this also makes the ordering relation more interesting.

Exercise 2. Specify this on paper, indicating (A) the set of lattice elements, (B) the relation between them, (c) the top element and (d) the bottom element.

Of course, this requires constituent changes in the flow function; we will outline a couple of examples on the board (but won't fully specify it; it gets a bit tedious!).

5.2 Constant Propagation

Constant propagation analysis attempts to track the constant values of variables in the program, where possible. Constant propagation has long been used in compiler optimization passes in order to turn variable reads and computations into constants. However, it is generally useful for analysis for program correctness as well: any client analysis that benefits from knowing program values (e.g. an array bounds analysis) can leverage it.

For constant propagation, we want to track what is the constant value, if any, of each program variable. Therefore we will use a lattice where the set L_{CP} is $\mathbb{Z} \cup \{\top, \bot\}$. The partial order is $\forall l \in L_{CP} : \bot \sqsubseteq l \wedge l \sqsubseteq \top$. In other words, $\bot$ is below every lattice element and $\top$ is above every element, but otherwise lattice elements are incomparable.

In the above lattice, as well as our earlier discussion of zero analysis, we used a lattice to describe individual variable values. We can lift the notion of a lattice to cover all the dataflow information available at a program point. This is called a *tuple lattice*, where there is an element of the tuple for each of the variables in the program. For constant propagation, the elements of the set σ are maps from *Var* to L_{CP}, and the other operators and $\top/\bot$ are lifted as follows:

$$\begin{array}{rcl} \sigma & \in & \mathit{Var} \to L_{CP} \\ \sigma_1 \sqsubseteq_{lift} \sigma_2 & \mathit{iff} & \forall x \in \mathit{Var} : \sigma_1(x) \sqsubseteq \sigma_2(x) \\ \sigma_1 \sqcup_{lift} \sigma_2 & = & \{x \mapsto \sigma_1(x) \sqcup \sigma_2(x) \mid x \in \mathit{Var}\} \\ \top_{lift} & = & \{x \mapsto \top \mid x \in \mathit{Var}\} \\ \bot_{lift} & = & \{x \mapsto \bot \mid x \in \mathit{Var}\} \end{array}$$

We can likewise define an abstraction function for constant propagation, as well as a lifted version that accepts an environment E mapping variables to concrete values. We also define the initial analysis information to conservatively assume that initial variable values are unknown. Note that in a language that initializes all variables to zero, we could make more precise initial dataflow assumptions, such as $\{x \mapsto 0 \mid x \in \mathit{Var}\}$:

$$\begin{array}{rcl} \alpha_{CP}(n) & = & n \\ \alpha_{lift}(E) & = & \{x \mapsto \alpha_{CP}(E(x)) \mid x \in \mathit{Var}\} \\ \sigma_0 & = & \top_{lift} \end{array}$$

We can now define flow functions for constant propagation:

$$\begin{array}{lll} f_{CP}[\![x := n]\!](\sigma) & = \sigma[x \mapsto \alpha_{CP}(n)] & \\ f_{CP}[\![x := y]\!](\sigma) & = \sigma[x \mapsto \sigma(y)] & \\ f_{CP}[\![x := y \mathit{\ op\ } z]\!](\sigma) & = \sigma[x \mapsto \sigma(y) \mathit{\ op}_{lift}\ \sigma(z)] & \\ & \text{where } n \mathit{\ op}_{lift}\ m = n \mathit{\ op\ } m & \\ & \quad \text{and } n \mathit{\ op}_{lift}\ \bot = \bot & \text{(and symmetric)} \\ & \quad \text{and } n \mathit{\ op}_{lift}\ \top = \top & \text{(and symmetric)} \\ f_{CP}[\![\text{goto } n]\!](\sigma) & = \sigma & \\ f_{CP}[\![\text{if } x = 0 \text{ goto } n]\!]_T(\sigma) & = \sigma[x \mapsto 0] & \\ f_{CP}[\![\text{if } x = 0 \text{ goto } n]\!]_F(\sigma) & = \sigma & \\ f_{CP}[\![\text{if } x < 0 \text{ goto } n]\!](\sigma) & = \sigma & \end{array}$$

We can now look at an example of constant propagation. Below, the code is on the left, and the results of the analysis is on the right. In this table we show the worklist as it is updated to show how the algorithm operates:

	stmt	worklist	x	y	z	w
1 : $x := 3$	0	1,2,3,4,5,6,7	$\top$	$\top$	$\top$	$\top$
2 : $y := x + 7$	1	2,3,4,5,6,7	3	$\top$	$\top$	$\top$
3 : if $z = 0$ goto 6	2	3,4,5,6,7	3	10	$\top$	$\top$
4 : $z := x + 2$	3	4,5,6,7	3	10	$0_T, \top_F$	$\top$
5 : goto 7	4	5,6,7	3	10	5	$\top$
6 : $z := y - 5$	5	6,7	3	10	5	$\top$
7 : $w := z - 2$	6	7	3	10	5	$\top$
	7	$\varnothing$	3	10	5	3

5.3 Reaching Definitions

Reaching definitions analysis determines, for each use of a variable, which assignments to that variable might have set the value seen at that use. Consider the following program:

$$
\begin{array}{ll}
1: & y := x \\
2: & z := 1 \\
3: & \text{if } y = 0 \text{ goto } 7 \\
4: & z := z * y \\
5: & y := y - 1 \\
6: & \text{goto } 3 \\
7: & y := 0
\end{array}
$$

In this example, definitions 1 and 5 reach the use of y at 4.

Exercise 3. Which definitions reach the use of z at statement 4?

Reaching definitions can be used as a simpler but less precise version of constant propagation, zero analysis, etc. where instead of tracking actual constant values we just look up the reaching definition and see if it is a constant. We can also use reaching definitions to identify uses of undefined variables, e.g. if no definition from the program reaches a use.

For reaching definitions, we define a new kind of lattice: a *set lattice*. Here, a dataflow lattice element is the set of definitions that reach the current program point. Assume that DEFS is the set of all definitions in the program. The set of elements in the lattice is the set of all subsets of DEFS—that is, the powerset of DEFS, written $\mathcal{P}^{\mathsf{DEFS}}$.

What should $\sqsubseteq$ be for reaching definitions? The intuition is that our analysis is more precise the *smaller* the set of definitions it computes at a given program point. This is because we want to know, as precisely as possible, where the values at a program point came from. So $\sqsubseteq$ should be the subset relation $\subseteq$: a subset is more precise than its superset. This naturally implies that $\sqcup$ should be $union$, and that $\top$ and $\bot$ should be the universal set DEFS and the empty set $\varnothing$, respectively.

In summary, we can formally define our lattice and initial dataflow information as follows:

$$
\begin{array}{rcl}
\sigma & \in & \mathcal{P}^{\mathsf{DEFS}} \\
\sigma_1 \sqsubseteq \sigma_2 & \textit{iff} & \sigma_1 \subseteq \sigma_2 \\
\sigma_1 \sqcup \sigma_2 & = & \sigma_1 \cup \sigma_2 \\
\top & = & \mathsf{DEFS} \\
\bot & = & \varnothing \\
\sigma_0 & = & \varnothing
\end{array}
$$

Instead of using the empty set for σ_0, we could use an artificial reaching definition for each program variable (e.g. x_0 as an artificial reaching definition for x) to denote that the variable is either uninitialized, or was passed in as a parameter. This is convenient if it is useful to track whether a variable might be uninitialized at a use, or if we want to consider a parameter to be a definition. We could write this formally as $\sigma_0 = \{x_0 \mid x \in \mathsf{Vars}\}$

We will now define flow functions for reaching definitions. Notationally, we will write x_n to denote a definition of the variable x at the program instruction numbered n. Since our lattice is a set, we can reason about changes to it in terms of elements that are added (called GEN) and elements that are removed (called KILL) for each statement. This GEN/KILL pattern is common to many dataflow analyses. The flow functions can be formally defined as follows:

$$
\begin{aligned}
f_{RD}[\![I]\!](\sigma) &= \sigma - KILL_{RD}[\![I]\!] \cup GEN_{RD}[\![I]\!] \\
\text{KILL}_{RD}[\![n\colon\ x := ...]\!] &= \{x_m \mid x_m \in \mathsf{DEFS}(x)\} \\
\text{KILL}_{RD}[\![I]\!] &= \varnothing \qquad \text{if } I \text{ is not an assignment} \\
\text{GEN}_{RD}[\![n\colon\ x := ...]\!] &= \{x_n\} \\
\text{GEN}_{RD}[\![I]\!] &= \varnothing \qquad \text{if } I \text{ is not an assignment}
\end{aligned}
$$

We would compute dataflow analysis information for the program shown above as follows (nodes that are added to the worklist again because of a change of dataflow values are depicted in **bold**):

stmt	worklist	defs
0	1,2,3,4,5,6,7	$\varnothing$
1	2,3,4,5,6,7	$\{y_1\}$
2	3,4,5,6,7	$\{y_1, z_1\}$
3	4,5,6,7	$\{y_1, z_1\}$
4	5,6,7	$\{y_1, z_4\}$
5	6,7	$\{y_5, z_4\}$
6	**3**,7	$\{y_5, z_4\}$
3	**4**,7	$\{y_1, y_5, z_1, z_4\}$
4	**5**,7	$\{y_1, y_5, z_4\}$
5	7	$\{y_5, z_4\}$
7	$\varnothing$	$\{y_7, z_1, z_4\}$

5.4 Live Variables

Live variable analysis determines, for each program point, which variables might be used again before they are redefined. Consider again the following program:

$$
\begin{aligned}
1:&\quad y := x \\
2:&\quad z := 1 \\
3:&\quad \text{if } y = 0 \text{ goto } 7 \\
4:&\quad z := z * y \\
5:&\quad y := y - 1 \\
6:&\quad \text{goto } 3 \\
7:&\quad y := 0
\end{aligned}
$$

In this example, after instruction 1, y is live, but x and z are not. Live variables analysis typically requires knowing what variable holds the main result(s) computed by the program. In the program above, suppose z is the result of the program. Then at the end of the program, only z is live.

Live variable analysis was originally developed for optimization purposes: if a variable is not live after it is defined, we can remove the definition instruction. For example, instruction 7 in the code above could be optimized away, under our assumption that z is the only program result of interest.

We must be careful of the side effects of a statement, of course. Assigning a variable that is no longer live to null could have the beneficial side effect of allowing the garbage collector to collect memory that is no longer reachable—unless the GC itself takes into consideration which variables are live. Sometimes warning the user that an assignment has no effect can be useful for software engineering purposes, even if the assignment cannot safely be optimized away.

For example, eBay found that FindBugs's analysis detecting assignments to dead variables was useful for identifying unnecessary database calls.[1]

For live variable analysis, we will use a set lattice to track the set of live variables at each program point. The lattice is similar to that for reaching definitions:

$$\begin{array}{rcl} \sigma & \in & \mathcal{P}^{\mathsf{Var}} \\ \sigma_1 \sqsubseteq \sigma_2 & \textit{iff} & \sigma_1 \subseteq \sigma_2 \\ \sigma_1 \sqcup \sigma_2 & = & \sigma_1 \cup \sigma_2 \\ \top & = & \mathsf{Var} \\ \bot & = & \varnothing \end{array}$$

What is the initial dataflow information? This is a tricky question. To determine the variables that are live at the start of the program, we must reason about how the program will execute...i.e. we must run the live variables analysis itself! There's no obvious assumption we can make about this. On the other hand, it is quite clear which variables are live at the *end* of the program: just the variable(s) holding the program result.

Consider how we might use this information to compute other live variables. Suppose the last statement in the program assigns the program result z, computing it based on some other variable x. Intuitively, that statement should make x live immediately above that statement, as it is needed to compute the program result z—but z should now no longer be live. We can use similar logic for the second-to-last statement, and so on. In fact, we can see that live variable analysis is a *backwards analysis*: we start with dataflow information at the *end* of the program and use flow functions to compute dataflow information at earlier statements.

Thus, for our "initial" dataflow information—and note that "initial" means the beginning of the program analysis, but the end of the program—we have:

$$\sigma_{end} = \{x \mid x \text{ holds part of the program result}\}$$

We can now define flow functions for live variable analysis. We can do this simply using GEN and KILL sets:

$$\begin{aligned} \text{KILL}_{LV}[\![I]\!] &= \{x \mid I \text{ defines } x\} \\ \text{GEN}_{LV}[\![I]\!] &= \{x \mid I \text{ uses } x\} \end{aligned}$$

We would compute dataflow analysis information for the program shown above as follows. Note that we iterate over the program backwords, i.e. reversing control flow edges between instructions. We also determine reverse postorder by considering the last statement–in this case line 7—as the "root". For each instruction, the corresponding row in our table will hold the information after we have applied the flow function—that is, the variables that are live immediately *before* the statement executes:

stmt	worklist	live
end	7,3,6,5,4,2,1	$\{z\}$
7	3,6,5,4,2,1	$\{z\}$
3	6,5,4,2,1	$\{z, y\}$
6	5,4,2,1	$\{z, y\}$
5	4,2,1	$\{z, y\}$
4	**3**,2,1	$\{z, y\}$
3	2	$\{z, y\}$
2	1	$\{y\}$
1	$\varnothing$	$\{x\}$

[1] see Ciera Jaspan, I-Chin Chen, and Anoop Sharma, *Understanding the value of program analysis tools*, OOPSLA practitioner report, 2007

Chapter 6

Dataflow Analysis Termination and Correctness

6.1 Termination

As we think about the correctness of program analysis, let us first think more carefully about the situations under which program analysis will terminate. In a previous lecture, we analyzed the performance of Kildall's worklist algorithm. A critical part of that performance analysis was the the observation that running a flow function on the same statement for the second time always either leaves the output dataflow analysis information unchanged, or makes it more approximate—that is, it moves the current dataflow analysis results up in the lattice, relative to the output when the flow function was run the first time. The dataflow values at each program point describe an *ascending chain*:

Ascending Chain A sequence σ_k is an *ascending chain* iff $n \leqslant m$ implies $\sigma_n \sqsubseteq \sigma_m$

We can define the height of an ascending chain, and of a lattice, in order to bound the number of new analysis values we can compute at each program point:

Height of an Ascending Chain An ascending chain σ_k has finite height h if it contains $h + 1$ distinct elements.

Height of a Lattice A lattice $(L, \sqsubseteq)$ has finite height h if there is an ascending chain in the lattice of height h, and no ascending chain in the lattice has height greater than h

We can now show that for a lattice of finite height, the worklist algorithm is guaranteed to terminate. We do so by showing that the dataflow analysis information at each program point follows an ascending chain. Consider again the worklist algorithm, this time in a slight variation that computes the new input to a node by joining the outputs of all its predecessors. This variation is in fact equivalent to the algorithm we saw before, but it will make our termination and correctness arguments more obvious. We assume a distinguished `programStart` node which comes before the first instruction:

```
worklist = ∅
for Node n in cfg
    input[n] = output[n] = ⊥
    add n to worklist
output[programStart] = initialDataflowInformation

while worklist is not empty
    take a Node n off the worklist
    input[n] = ⊔_{k∈preds(n)} output[k]
    newOutput = flow(n, input[n])
    if newOutput ≠ output[n]
        output[n] = newOutput
        for Node j in succs(n)
            add j to worklist
```

We can make an intuitive inductive argument for termination: At the beginning of the analysis, the analysis information before and after every program point (other than after the program start node) is $\bot$ (by definition). Thus the first time we run each flow function for each instruction, the result will be at least as high in the lattice as what was there before (because nothing is lower in a lattice than $\bot$). We will run the flow function for a given instruction again at a program point only if the output from a predecessor instruction changes. Assume that the previous time we ran the flow function, we had input information σ_i and output information σ_o. Now we are running it again because the input dataflow analysis information has changed to some new σ_i'—and by the induction hypothesis, we can assume it is higher in the lattice than before, i.e. $\sigma_i \sqsubseteq \sigma_i'$.

What we need to show is that the output information σ_o' is at least as high in the lattice as the old output information σ_o—that is, we must show that $\sigma_o \sqsubseteq \sigma_o'$. This will be true if our flow functions are monotonic:

Monotonicity A function f is *monotonic* iff $\sigma_1 \sqsubseteq \sigma_2$ implies $f(\sigma_1) \sqsubseteq f(\sigma_2)$

Now we can state the termination theorem:

Theorem 1 (Dataflow Analysis Termination)**.** *If a dataflow lattice $(L, \sqsubseteq)$ has finite height, and the corresponding flow functions are monotonic, the worklist algorithm will terminate.*

Proof. The idea should be intuitively clear from the argument above. However, to make it rigorous, we provide the following termination metric:

$$M = |worklist| + EpN * LC(\sigma)$$

where $|worklist|$ is the length of the worklist, EpN is the maximum number of outgoing *Edges per Node*, and $LC(\sigma)$ is the longest aescending chain from σ to $\top$. When computing $LC(\sigma)$ we consider σ to be one big lattice, i.e. a tuple constructed from the sub-lattices for each program point, so that moving up in the sub-lattice for any program point moves the overall σ lattice up as well.

M is finite because $|worklist|$ is bounded by the number of nodes in the program, EpN is finite, and the lattice σ is of finite height (which we know because it is a tuple lattice with a finite number of sub-lattices, all of which have finite height by the assumption in the theorem).

M decreases on each iteration of the loop, as follows. $|worklist|$ generally decreases by one in each iteration because one node is removed from it. However, we must account for additions to the worklist when the `newOutput ≠ output[i]` condition holds. But note that when this condition holds, `newOutput` must be higher in the lattice than `output[i]` by monotonicity. Thus, running the flow function reduced $LC(\sigma)$ by at least one. We then add

at most EpN nodes to the worklist. The increase to the worklist is at least balanced by the decrease in $EpN * LC(\sigma)$. Thus, the metric M decreases even when the condition that results in adding nodes to the worklist holds.

Exercise 1. Convince yourself that, for monotonic flow functions, the algorithm above does the same thing as the algorithm given in a previous lecture.

□

6.2 Montonicity of Zero Analysis

We can formally show that zero analysis is monotone; this is relevant both to the proof of termination, above, and to correctness, next. We will only give a couple of the more interesting cases, and leave the rest as an exercise to the reader:

Case $f_Z[\![x := 0]\!](\sigma) = \sigma[x \mapsto Z]$:
Assume we have $\sigma_1 \sqsubseteq \sigma_2$
Since $\sqsubseteq$ is defined pointwise, we know that $\sigma_1[x \mapsto Z] \sqsubseteq \sigma_2[x \mapsto Z]$

Case $f_Z[\![x := y]\!](\sigma) = \sigma[x \mapsto \sigma(y)]$:
Assume we have $\sigma_1 \sqsubseteq \sigma_2$
Since $\sqsubseteq$ is defined pointwise, we know that $\sigma_1(y) \sqsubseteq_{simple} \sigma_2(y)$
Therefore, using the pointwise definition of $\sqsubseteq$ again, we also obtain $\sigma_1[x \mapsto \sigma_1(y)] \sqsubseteq \sigma_2[x \mapsto \sigma_2(y)]$

(α_{simple} and $\sqsubseteq_{simple}$ are simply the unlifted versions of α and $\sqsubseteq$, i.e. they operate on individual values rather than maps.)

Exercise 2. Write an alternative (incorrect) flow function for zero analysis that is non-monotone, and write a program on which dataflow analysis using your alternative flow function will not terminate.

6.3 Correctness

What does it mean for an analysis of a WHILE3ADDR program to be correct? Intuitively, we would like the program analysis results to correctly describe every actual execution of the program. To establish correctness, we will make use of the precise definitions of WHILE3ADDR we gave in the form of operational semantics in the first couple of lectures. We start by formalizing a program execution as a trace:

Program Trace A trace T of a program P is a potentially infinite sequence $\{c_0, c_1, ...\}$ of program configurations, where $c_0 = E_0, 1$ is called the initial configuration, and for every $i \geqslant 0$ we have $P \vdash c_i \rightsquigarrow c_{i+1}$

Given this definition, we can formally define soundness:

Dataflow Analysis Soundness The result $\{\sigma_n \mid n \in P\}$ of a program analysis running on program P is sound iff, for all traces T of P, for all i such that $0 \leqslant i < \mathit{length}(T)$, $\alpha(c_i) \sqsubseteq \sigma_{n_i}$

In this definition, just as c_i is the program configuration immediately before executing instruction n_i as the ith program step, σ_{n_i} is the dataflow analysis information immediately before instruction n_i.

Exercise 3. Consider the following (incorrect) flow function for zero analysis:

$$f_Z[\![x := y + z]\!](\sigma) = \sigma[x \mapsto Z]$$

Give an example of a program and a concrete trace that illustrates that this flow function is unsound.

The key to designing a sound analysis is to make sure that the flow functions map abstract information before each instruction to abstract information after that instruction in a way that matches the instruction's concrete semantics. Another way of saying this is that the manipulation of the abstract state done by the analysis should reflect the manipulation of the concrete machine state done by the executing instruction. We can formalize this as a *local soundness* property:

Local Soundness A flow function f is *locally sound* iff $P \vdash c_i \rightsquigarrow c_{i+1}$ and $\alpha(c_i) \sqsubseteq \sigma_{n_i}$ and $f[\![P[n_i]]\!](\sigma_{n_i}) = \sigma_{n_{i+1}}$ implies $\alpha(c_{i+1}) \sqsubseteq \sigma_{n_{i+1}}$

In English: if we take any concrete execution of a program instruction, map the input machine state to the abstract domain using the abstraction function, find that the abstracted input state is described by the analysis input information, and apply the flow function, we should get a result that correctly accounts for what happens if we map the actual concrete output machine state to the abstract domain.

Exercise 4. Consider again the incorrect zero analysis flow function described above. Specify an input state c_i and use that input state to show that the flow function is not locally sound.

We can now show that the flow functions for zero analysis are locally sound. Although technically the overall abstraction function α accepts a complete program configuration (E, n), for zero analysis we can ignore the n component and so in the proof below we will simply focus on the environment E. We show the cases for a couple of interesting syntax forms; the rest are either trivial or analogous:

Case $f_Z[\![x := 0]\!](\sigma_{n_i}) = \sigma_{n_i}[x \mapsto Z]$:
Assume $c_i = E, n$ and $\alpha(E) \sqsubseteq \sigma_{n_i}$
Thus $\sigma_{n_{i+1}} = f_Z[\![x := 0]\!](\sigma_{n_i}) = \sigma_{n_i}[x \mapsto Z]$
$c_{i+1} = E[x \mapsto 0], n + 1$ by rule *step-const*
Now $\alpha(c_{i+1}) = \alpha(E[x \mapsto 0]) = \alpha(E)[x \mapsto Z]$ by the definition of α.
$\alpha(E) \sqsubseteq \sigma_{n_i}$ implies $\alpha(c_{i+1}) = \alpha(E)[x \mapsto Z] \sqsubseteq \sigma_{n_i}[x \mapsto Z] = \sigma_{n_{i+1}}$,
so therefore $\alpha(c_{i+1}) \sqsubseteq \sigma_{n_{i+1}}$, which finishes the case.

Case $f_Z[\![x := m]\!](\sigma_{n_i}) = \sigma_{n_i}[x \mapsto N]$ where $m \neq 0$:
Assume $c_i = E, n$ and $\alpha(E) \sqsubseteq \sigma_{n_i}$
Thus $\sigma_{n_{i+1}} = f_Z[\![x := m]\!](\sigma_{n_i}) = \sigma_{n_i}[x \mapsto N]$
$c_{i+1} = E[x \mapsto m], n + 1$ by rule *step-const*
Now $\alpha(c_{i+1}) = \alpha(E[x \mapsto m]) = \alpha(E)[x \mapsto N]$ by the definition of α and the assumption that $m \neq 0$.
$\alpha(E) \sqsubseteq \sigma_{n_i}$ implies $\alpha(c_{i+1}) = \alpha(E)[x \mapsto N] \sqsubseteq \sigma_{n_i}[x \mapsto N] = \sigma_{n_{i+1}}$.
so therefore $\alpha(c_{i+1}) \sqsubseteq \sigma_{n_{i+1}}$ which finishes the case.

Case $f_Z[\![x := y \ op \ z]\!](\sigma_{n_i}) = \sigma_{n_i}[x \mapsto \top]$:
Assume $c_i = E, n$ and $\alpha(E) \sqsubseteq \sigma_{n_i}$
Thus $\sigma_{n_{i+1}} = f_Z[\![x := y \ op \ z]\!](\sigma_{n_i}) = \sigma_{n_i}[x \mapsto \top]$
$c_{i+1} = E[x \mapsto k], n+1$ for some k by rule *step-arith*
Now $\alpha(c_{i+1}) = \alpha(E[x \mapsto k]) \sqsubseteq \alpha(E)[x \mapsto \top]$ because the map is equal for all keys except x, and for x we have $\alpha_{simple}(k) \sqsubseteq_{simple} \top$ for all k, where α_{simple} and $\sqsubseteq_{simple}$ are the unlifted versions of α and $\sqsubseteq$, i.e. they operate on individual values rather than maps.
$\alpha(E) \sqsubseteq \sigma_{n_i}$ implies $\alpha(c_{i+1}) = \alpha(E[x \mapsto k]) \sqsubseteq \alpha(E)[x \mapsto \top] \sqsubseteq \sigma_{n_i}[x \mapsto \top] = \sigma_{n_{i+1}}$, so therefore, by transitivity of $\sqsubseteq$, $\alpha(c_{i+1}) \sqsubseteq \sigma_{n_{i+1}}$ which finishes the case.

Exercise 5. Prove the case for $f_Z[\![x := y]\!](\sigma) = \sigma[x \mapsto \sigma(y)]$.

Now we can show that local soundness can be used to prove the global soundness of a dataflow analysis. To do so, let us formally define the state of the dataflow analysis at a fixed point:

Fixed Point	A dataflow analysis result $\{\sigma_i \mid i \in P\}$ is a fixed point iff $\sigma_0 \sqsubseteq \sigma_1$ where σ_0 is the initial analysis information and σ_1 is the information before the first instruction, and for each instruction i we have $\bigsqcup_{j \in \texttt{preds}(i)} f[\![P[j]]\!](\sigma_j) \sqsubseteq \sigma_i$.

The worklist algorithm show above computes a fixed point when it terminates. We can prove this by showing that the following loop invariant is maintained:

$$\forall i \,.\, (\exists j \in \mathit{preds}(i) \textit{ such that } f[\![P[j]]\!](\sigma_j) \not\sqsubseteq \sigma_i) \Rightarrow i \in \texttt{worklist}$$

The invariant is trivially true initially because all instructions are initially in the worklist. The invariant is maintained because whenever the output $f[\![P[j]]\!](\sigma_j)$ of instruction j changes, possibly breaking the invariant, then the successors of j are added to the worklist, thus restoring it. When an instruction i is removed from the worklist and processed, the invariant as it applies to i is established. Finally, when the worklist is empty, the definition above is equivalent to the definition of a fixed point.

And now the main result we will use to prove program analyses correct:

Theorem 2 (A fixed point of a locally sound analysis is globally sound)**.** *If a dataflow analysis's flow function f is monotonic and locally sound, and for all traces T we have $\alpha(c_0) \sqsubseteq \sigma_0$ where σ_0 is the initial analysis information, then any fixed point $\{\sigma_n \mid n \in P\}$ of the analysis is sound.*

Proof. To show that the analysis is sound, we must prove that for all program traces, every program configuration in that trace is correctly approximated by the analysis results. We consider an arbitrary program trace T and do the proof by induction on the program configurations $\{c_i\}$ in the trace.

Case c_0:

$\alpha(c_0) \sqsubseteq \sigma_0$ by assumption.
$\sigma_0 \sqsubseteq \sigma_{n_0}$ by the definition of a fixed point.
$\alpha(c_0) \sqsubseteq \sigma_{n_0}$ by the transitivity of $\sqsubseteq$.

Case c_{i+1}:

$\alpha(c_i) \sqsubseteq \sigma_{n_i}$ by the induction hypothesis.
$P \vdash c_i \rightsquigarrow c_{i+1}$ by the definition of a trace.
$\alpha(c_{i+1}) \sqsubseteq f[\![P[n_i]]\!](\sigma_{n_i})$ by local soundness.
$f[\![P[n_i]]\!](\sigma_{n_i}) \sqcup ... \sqsubseteq \sigma_{n_{i+1}}$ by the definition of fixed point.
$f[\![P[n_i]]\!](\sigma_{n_i}) \sqsubseteq \sigma_{n_{i+1}}$ by the properties of $\sqcup$.
$\alpha(c_{i+1}) \sqsubseteq \sigma_{n_{i+1}}$ by the transitivity of $\sqsubseteq$.

□

Since we previously proved that Zero Analysis is locally sound and that its flow functions are monotonic, we can use this theorem to conclude that the analysis is sound. This means, for example, that Zero Analysis will never neglect to warn us if we are dividing by a variable that could be zero.

Chapter 7

Widening Operators and Collecting Semantics for Dataflow Analysis

The approaches for proving termination and correctness outlined in the previous chapter rely on certain assumptions that do not hold for all static analyses. In this chapter, we show examples of how to relax those assumptions via widening operators and collecting semantics, and then overview how they are tied together via the Abstract Interpretation framework.

7.1 Widening operators: Dealing with Infinite-Height Lattices

Both our informal intuition and the formal proof of termination we have discussed so far rely on an abstract domain L that contains no infinite ascending chains. However, there are certainly applications where such abstractions would be useful! How can we develop useful, terminating analyses in such cases?

7.1.1 Example: Interval Analysis

Let us consider a program analysis that might be suitable for array bounds checking, namely *interval analysis*. As the name suggests, interval analysis tracks the interval of values that each variable might hold. We can define a lattice, initial dataflow information, and abstraction function as follows:

$$\begin{array}{rcl} L & = & \mathbb{Z}_\infty \times \mathbb{Z}_\infty \qquad \text{where } \mathbb{Z}_\infty = \mathbb{Z} \cup \{-\infty, \infty\} \\ [l_1, h_1] \sqsubseteq [l_2, h_2] & \textit{iff} & l_2 \leqslant_\infty l_1 \wedge h_1 \leqslant_\infty h_2 \\ [l_1, h_1] \sqcup [l_2, h_2] & = & [min_\infty(l_1, l_2), max_\infty(h_1, h_2)] \\ \top & = & [-\infty, \infty] \\ \bot & = & [\infty, -\infty] \\ \sigma_0 & = & \top \\ \alpha(x) & = & [x, x] \end{array}$$

We have extended the $\leqslant$ operator and the *min* and *max* functions to handle sentinels representing positive and negative infinity in the obvious way. For example $-\infty \leqslant_\infty n$ for all $n \in \mathbb{Z}$. For convenience we write the empty interval $\bot$ as $[\infty, -\infty]$.

Note also that this lattice is defined to capture the range of a single variable. As usual, we can lift it to a map from variables to interval lattice elements. Thus we (again) have dataflow information $\sigma \in \mathbf{Var} \to L$

We can also define a set of flow functions. Here we provide one for addition; the rest should be easy for the reader to develop:

$$f_I[\![x := y + z]\!](\sigma) = \sigma[x \mapsto [l, h]] \quad \text{where } l = \sigma(y).low +_\infty \sigma(z).low \text{ and } h = \sigma(y).high +_\infty \sigma(z).high$$

$$f_I[\![x := y + z]\!](\sigma) = \sigma \quad \text{where } \sigma(y) = \bot \vee \sigma(z) = \bot$$

In the above we have extended mathematical $+$ to operate over the sentinels for $\infty, -\infty$, for example such that $\forall x \neq -\infty : \infty + x = \infty$. We define the second case of the flow function to handle the case where one argument is $\bot$, possibly resulting in the undefined case $-\infty + \infty$.

If we run this analysis on a program, whenever we come to an array dereference, we can check whether the interval produced by the analysis for the array index variable is within the bounds of the array. If not, we can issue a warning about a potential array bounds violation.

Just one practical problem remains. Consider: *what is the height of the above-defined lattice, and what consequences does this have for our analysis in practice?*

7.1.2 The Widening Operator

As in the example of interval analysis, there are times in which it is useful to define a lattice of infinite height. We would like to nevertheless find a mechanism for ensuring that the analysis will terminate. One way to do this is to find situations where the lattice may be ascending an infinite chain at a given program point, and effectively shorten the chain to a finite height. We can do so with a *widening operator*. To motivate the widening operator, consider applying interval analysis to the program below:

```
1 :  x := 0
2 :  if x = y goto 5
3 :  x := x + 1
4 :  goto 2
5 :  y := 0
```

Using the worklist algorithm (strongly connected components first), gives us:

stmt	worklist	x	y
0	1,2,3,4,5	$\top$	$\top$
1	2,3,4,5	[0,0]	$\top$
2	3,4,5	[0,0]	$\top$
3	4,5	[1,1]	$\top$
4	2,5	[1,1]	$\top$
2	3,5	[0,1]	$\top$
3	4,5	[1,2]	$\top$
4	2,5	[1,2]	$\top$
2	3,5	[0,2]	$\top$
3	4,5	[1,3]	$\top$
4	2,5	[1,3]	$\top$
2	3,5	[0,3]	$\top$
...			

Consider the sequence of interval lattice elements for x immediately after statement 2. Counting the original lattice value as $\bot$ (not shown explicitly in the trace above), we can see it is the ascending chain $\bot, [0,0], [0,1], [0,2], [0,3], ...$ Recall that ascending chain means that each element of the sequence is higher in the lattice than the previous element. In the case of interval analysis, [0,2] (for example) is higher than [0,1] in the lattice because the latter interval is contained within the former. Given mathematical integers, this chain is clearly infinite; therefore our analysis is not guaranteed to terminate (and indeed it will not in practice).

A widening operator's purpose is to compress such infinite chains to finite length. The widening operator considers the most recent two elements in a chain. If the second is higher than the first, the widening operator can choose to jump up in the lattice, potentially skipping elements in the chain. For example, one way to cut the ascending chain above down to a finite height is to observe that the upper limit for x is increasing, and therefore assume the maximum possible value ∞ for x. Thus we will have the new chain $\bot, [0,0], [0,\infty], [0,\infty], ...$ which has already converged after the third element in the sequence.

The widening operator gets its name because it is an upper bound operator, and in many lattices, higher elements represent a wider range of program values.

We can define the example widening operator given above more formally as follows:

$$
\begin{array}{lll}
W(\bot, l_{current}) & = l_{current} & \\
W([l_1,h_1],[l_2,h_2]) & = [min_W(l_1,l_2), max_W(h_1,h_2)] & \\
 & \text{where } min_W(l_1,l_2) = l_1 & \text{if } l_1 \leqslant l_2 \\
 & \text{and } min_W(l_1,l_2) = -\infty & \text{otherwise} \\
 & \text{where } max_W(h_1,h_2) = h_1 & \text{if } h_1 \geqslant h_2 \\
 & \text{and } max_W(h_1,h_2) = \infty & \text{otherwise}
\end{array}
$$

Applying this widening operator each time just before analyzing instruction 2 produces:

stmt	worklist	x	y
0	1,2,3,4,5	$\top$	$\top$
1	2,3,4,5	[0,0]	$\top$
2	3,4,5	[0,0]	$\top$
3	4,5	[1,1]	$\top$
4	2,5	[1,1]	$\top$
2	3,5	[0,∞]	$\top$
3	4,5	[1,∞]	$\top$
4	2,5	[1,∞]	$\top$
2	5	[0,∞]	$\top$
5	$\varnothing$	[0,∞]	[0,0]

Before we analyze instruction 2 the first time, we compute $W(\bot, [0,0]) = [0,0]$ using the first case of the definition of W. Before we analyze instruction 2 the second time, we compute $W([0,0],[0,1]) = [0,\infty]$. In particular, the lower bound 0 has not changed, but since the upper bound has increased from $h_1 = 0$ to $h_2 = 1$, the max_W helper function sets the maximum to ∞. After we go through the loop a second time we observe that iteration has converged at a fixed point. We therefore analyze statement 5 and we are done.

Let us consider the properties of widening operators more generally. A widening operator $W(l_{previous}{:}L,\ l_{current}{:}L) : L$ accepts two lattice elements, the previous lattice value $l_{previous}$ at a program location and the current lattice value $l_{current}$ at the same program location. It returns a new lattice value that will be used in place of the current lattice value.

We require two properties of widening operators. The first is that the widening operator must return an upper bound of its operands. Intuitively, this is required for monotonicity: if the operator is applied to an ascending chain, the result should also be an ascending chain. Formally, we have $\forall l_{previous}, l_{current} : l_{previous} \sqsubseteq W(l_{previous}, l_{current}) \wedge l_{current} \sqsubseteq W(l_{previous}, l_{current})$.

The second property is that when the widening operator is applied to an ascending chain l_i, the resulting ascending chain l_i^W must be of finite height. Formally we define $l_0^W = l_0$ and $\forall i > 0 : l_i^W = W(l_{i-1}^W, l_i)$. This property ensures that when we apply the widening operator, it will ensure that the analysis terminates.

Where can we apply the widening operator? Clearly it is safe to apply anywhere, since it must be an upper bound and therefore can only raise the analysis result in the lattice, thus making the analysis result more conservative. However, widening inherently causes a loss of precision. Therefore it is better to apply it only when necessary. One solution is to apply the widening operator only at the heads of loops, as in the example above. Loop heads (or their equivalent, in unstructured control flow) can be inferred even from low-level three address code—see a compiler text such as Appel and Palsberg's *Modern Compiler Implementation in Java*.

We can use a somewhat smarter version of this widening operator with the insight that the bounds of a lattice are often related to constants in the program. Thus if we have an ascending chain $\bot, [0,0], [0,1], [0,2], [0,3], ...$ and the constant 10 is in the program, we might change the chain to $\bot, [0,0], [0,10], ...$ If we are lucky, the chain will stop ascending that that point: $\bot, [0,0], [0,10], [0,10], ...$ If we are not so fortunate, the chain will continue and eventually stabilize at $[0,\infty]$ as before: $\bot, [0,0], [0,10], [0,\infty]$.

If the program has the set of constants K, we can define a widening operator as follows:

$$
\begin{array}{lll}
W(\bot, l_{current}) & = l_{current} & \\
W([l_1,h_1],[l_2,h_2]) & = [min_K(l_1,l_2), max_K(h_1,h_2)] & \\
 & \text{where } min_K(l_1,l_2) = l_1 & \text{if } l_1 \leqslant l_2 \\
 & \text{and } min_K(l_1,l_2) = max(\{k \in K | k \leqslant l_2\}) & \text{otherwise} \\
 & \text{where } max_K(h_1,h_2) = h_1 & \text{if } h_1 \geqslant h_2 \\
 & \text{and } max_K(h_1,h_2) = min(\{k \in K | k \geqslant h_2\} & \text{otherwise}
\end{array}
$$

We can now analyze a program with a couple of constants and see how this approach works:

$$
\begin{array}{ll}
1: & x := 0 \\
2: & y := 1 \\
3: & \text{if } x = 10 \text{ goto } 7 \\
4: & x := x + 1 \\
5: & y := y - 1 \\
6: & \text{goto } 3 \\
7: & \text{goto } 7
\end{array}
$$

Here the constants in the program are $0, 1$ and 10. The analysis results are as follows:

stmt	worklist	x	y
0	1,2,3,4,5,6,7	$\top$	$\top$
1	2,3,4,5,6,7	[0,0]	$\top$
2	3,4,5,6,7	[0,0]	$[1,1]$
3	4,5,6,7	$[0,0]_F, \bot_T$	$[1,1]$
4	5,6,7	[1,1]	$[1,1]$
5	6,7	[1,1]	$[0,0]$
6	3,7	[1,1]	$[0,0]$
3	4,7	$[0,1]_F, \bot_T$	$[0,1]$
4	5,7	[1,2]	$[0,1]$
5	6,7	[1,2]	$[-1,0]$
6	3,7	[1,2]	$[-1,0]$
3	4,7	$[0,9]_F, [10,10]_T$	$[-\infty,1]$
4	5,7	[1,10]	$[-\infty,1]$
5	6,7	[1,10]	$[-\infty,0]$
6	3,7	[1,10]	$[-\infty,0]$
3	7	$[0,9]_F, [10,10]_T$	$[-\infty,1]$
7	$\varnothing$	[10,10]	$[-\infty,1]$

Applying the widening operation the first time we get to statement 3 has no effect, as the previous analysis value was $\bot$. The second time we get to statement 3, the range of both x and y has been extended, but both are still bounded by constants in the program. The third time we get to statement 3, we apply the widening operator to x, whose abstract value has gone from [0,1] to [0,2]. The widened abstract value is [0,10], since 10 is the smallest constant in the program that is at least as large as 2. For y we must widen to $[-\infty, 1]$. The analysis stabilizes after one more iteration.

In this example I have assumed a flow function for the if instruction that propagates different interval information depending on whether the branch is taken or not. In the table, we list the branch taken information for x as $\bot$ until x reaches the range in which it is feasible to take the branch. $\bot$ can be seen as a natural representation for dataflow values that propagate along a path that is infeasible.

7.2 Collecting Semantics (Reaching Definitions)

The approach to dataflow analysis correctness outlined in the previous lectures generalizes naturally when we have a lattice that can be directly abstracted from program configurations c from our execution semantics. Sometimes, however, it would be useful to track other kinds of information, that we cannot get directly from a particular state in program execution.

Consider *reaching definitions*, which we discussed as an example analysis in previous chapters. Although we can track which definitions reach a line using the previously-outlined approach, we cannot see *where* the variables used in an instruction I were last defined. The direct mapping that α provides between concrete and abstract states we used in our proof of correctness of zero analysis does not hold here.

To solve this problem, we can augment our semantics with additional information that captures the required information. For example, for reaching definitions, we want to know, at any point in a particular execution, which *definition* reaches the current location for each program variable in scope.

We call a version of the program semantics that has been augmented with additional information necessary for some particular analysis a *collecting semantics*. For reaching definitions, we can define a collecting semantics with a version of the environment E, which we will call

E_{RD}, that has been extended with a index n indicating the location where each variable was last defined.

$$E_{RD} \quad \in \quad \mathit{Var} \rightarrow \mathbb{Z} \times \mathbb{N}$$

We can now extend the semantics to track this information. We show only the rules that differ from those described in the earlier lectures:

$$\frac{P(n) = x := m}{P \vdash \langle E, n\rangle \rightsquigarrow \langle E[x \mapsto m, n], n+1\rangle} \textit{ step-const}$$

$$\frac{P(n) = x := y}{P \vdash \langle E, n\rangle \rightsquigarrow \langle E[x \mapsto E(y), n], n+1\rangle} \textit{ step-copy}$$

$$\frac{P(n) = x := y \mathit{\ op\ } z \quad E(y) \mathbf{\ op\ } E(z) = m}{P \vdash \langle E, n\rangle \rightsquigarrow \langle E[x \mapsto m, n], n+1\rangle} \textit{ step-arith}$$

Essentially, each rule that defines a variable records the current location as the latest definition of that variable. Now we can define an abstraction function for reaching definitions from this collecting semantics:

$$\alpha_{RD}(E_{RD}, n) = \{x_m \mid \exists x \in \mathit{domain}(E_{RD}) \text{ such that } E_{RD}(x) = i, m\}$$

From this point, reasoning about the correctness of reaching definitions proceeds analogously to the reasoning for zero analysis outlined in the previous lectures.

Formulating a collecting semantics can be tricky for some analyses, but it can be done with a little thought. For example, consider live variable analysis. The collecting semantics requires us to know, for each execution of the program, which variables currently in scope will be used before they are defined in the remainder of the program. We can compute this semantics by assuming a (possibly infinite) trace for a program run, then specifying the set of live variables at every point in that trace based on the trace going forward from that point. This semantics, specified in terms of traces rather than a set of inference rules, can then be used in the definition of an abstraction function and used to reason about the correctness of live variables analysis.

Chapter 8

Interprocedural Analysis

Consider an extension of WHILE3ADDR that includes functions. We thus add a new syntactic category F (for functions), and two new instruction forms (function call and return), as follows:

$$\begin{array}{rcl} F & ::= & \text{fun } f(x) \{ \overline{n : I} \} \\ I & ::= & \ldots \mid \text{return } x \mid y := f(x) \end{array}$$

In the notation above, $\overline{n : I}$, the line is shorthand for a list, so that the body of a function is a list of instructions I with line numbers n. We assume in our formalism that all functions take a single integer argument and return an integer result, but this is easy to generalize if we need to. We can also add global variables to this language by tracking a separate set of variables, **Globals**. We assume simple syntactic scoping.

Note that this is not a truly precise syntactic specification. Specifying even just "possibly empty list of arithmetic expressions" properly takes several intermediate syntactic steps; correctly handling scope requires rather significant refinement to the operational semantics. However, providing such precision is more trouble than it's worth for this discussion. Function names are strings. Functions may return either void or a single integer. We leave the problem of type-checking to another class.

We've made our programming language much easier to use, but dataflow analysis has become rather more difficult. Interprocedural analysis concerns analyzing a program with multiple procedures, ideally taking into account the way that information flows among those procedures. We use zero analysis as our running example throughout, unless otherwise indicated.

8.1 Two Simple Approaches

Default assumptions. Our first approach assumes a default lattice value for all arguments to a function L_a and a default value for procedure results L_r. In some respects, L_a is equivalent to the initial dataflow information we set at the entry to the program when we were only looking intraprocedurally; now we assume it on entry to every procedure. We check the assumptions hold when analyzing a call or return instruction (trivial if $L_a = L_r = \top$). We then use the assumption when analyzing the result of a call instruction or starting the analysis of a method. For example, we have $\sigma_0 = \{x \mapsto L_a \mid x \in \mathbf{Var}\}$.

Here is a sample flow function for call and return instructions:

$$\begin{array}{lll} f[\![x := g(y)]\!](\sigma) & = \sigma[x \mapsto L_r] & (\text{error if } \sigma(y) \not\sqsubseteq L_a) \\ f[\![\text{return } x]\!](\sigma) & = \sigma & (\text{error if } \sigma(x) \not\sqsubseteq L_r) \end{array}$$

We can apply zero analysis to the following function, using $L_a = L_r = \top$:

$$
\begin{array}{ll}
1: & \text{fun } \textit{divByX}(x) : int \\
2: & \quad y := 10/x \\
3: & \quad \text{return } y \\
4: & \text{fun } \textit{main}() : void \\
5: & \quad z := 5 \\
6: & \quad w := \textit{divByX}(z)
\end{array}
$$

The results are sound, but imprecise. We can avoid the false positive by using a more optimistic assumption $L_a = L_r = NZ$. But then we get a problem with the following program:

$$
\begin{array}{ll}
1: & \text{fun } \textit{double}(x) : int \\
2: & \quad y := 2 * x \\
3: & \quad \text{return } y \\
4: & \text{fun } \textit{main}() : void \\
5: & \quad z := 0 \\
6: & \quad w := \textit{double}(z)
\end{array}
$$

Now what?

Annotations. An alternative approach uses *annotations*. This allows us to choose different argument and result assumptions for different procedures. Flow functions might look like:

$$
\begin{array}{lll}
f[\![x := g(y)]\!](\sigma) & = \sigma[x \mapsto \textit{annot}[\![g]\!].r] & (\text{error if } \sigma(y) \not\sqsubseteq \textit{annot}[\![g]\!].a) \\
f[\![\text{return } x]\!](\sigma) & = \sigma & (\text{error if } \sigma(x) \not\sqsubseteq \textit{annot}[\![g]\!].r)
\end{array}
$$

Now we can verify that both of the above programs are safe, given the proper annotations. We will see other example analysis approaches that use annotations later in the semester, though historically, programmer buy-in remains a challenge in practice.

Local vs. global variables. If we add global variables, we must make conservative assumptions about them too. Assume globals should always be described by some lattice value L_g at procedure boundaries. We can extend the flow functions as follows:

$$
\begin{array}{lll}
f[\![x := g(y)]\!](\sigma) & = \sigma[x \mapsto L_r][z \mapsto L_g \mid z \in \textbf{Globals}] \\
& \quad (\text{error if } \sigma(y) \not\sqsubseteq L_a \vee \forall z \in \textbf{Globals} : \sigma(z) \not\sqsubseteq L_g) \\
f[\![\text{return } x]\!](\sigma) & = \sigma \\
& \quad (\text{error if } \sigma(x) \not\sqsubseteq L_r \vee \forall z \in \textbf{Globals} : \sigma(z) \not\sqsubseteq L_g)
\end{array}
$$

The annotation approach can also be extended in a natural way to handle global variables.

8.2 Interprocedural Control Flow Graphs

An approach that avoids the burden of annotations, and can capture what a procedure actually does as used in a particular program, is to build a control flow graph for the entire program, rather than just a single procedure. To make this work, we handle call and return instructions specially as follows:

- We add additional edges to the control flow graph. For every call to function g, we add an edge from the call site to the first instruction of g, and from every return statement of g to the instruction following that call.

- When analyzing the first statement of a procedure, we generally gather analysis information from each predecessor as usual. However, we take out all dataflow information related to local variables in the callers. Furthermore, we add dataflow information for parameters in the callee, initializing their dataflow values according to the actual arguments passed in at each call site.
- When analyzing an instruction immediately after a call, we get dataflow information about local variables from the previous statement. Information about global variables is taken from the return sites of the function that was called. Information about the variable that the result of the function call was assigned to comes from the dataflow information about the returned value.

Now the examples described above can be successfully analyzed. However, other programs still cause problems:

$$
\begin{array}{ll}
1: & \text{fun } \mathit{double}(x) : \mathit{int} \\
2: & \quad y := 2 * x \\
3: & \quad \text{return } y \\
4: & \text{fun } \mathit{main}() \\
5: & \quad z := 5 \\
6: & \quad w := \mathit{double}(z) \\
7: & \quad z := 10/w \\
8: & \quad z := 0 \\
9: & \quad w := \mathit{double}(z)
\end{array}
$$

What's the issue here?

8.3 Context Sensitive Analysis

Context-sensitive analysis analyzes a function either multiple times, or parametrically, so that the analysis results returned to different call sites reflect the different analysis results passed in at those call sites. We could get context sensitivity just by duplicating (or inlining) all callees, but this only works for non-recursive programs.

A simple solution is to build a summary of each function, mapping dataflow input information to dataflow output information. We will analyze each function once for each *context*, where a context is an abstraction for a set of calls to that function. At a minimum, each context must track the input dataflow information to the function.

Let's look at how this approach allows the program given above to be proven safe by zero analysis...*(Example will be given in class)*

Things become more challenging in the presence of recursive functions, or more generally mutual recursion. Let us consider context-sensitive interprocedural constant propagation analysis of a factorial function called by main. We are not focused on the intraprocedural part of the analysis, so we will just show the function in the form of Java or C source code:

```
int fact(int x) {
    if (x == 1)
        return 1;
    else
        return x * fact(x-1);
}
```

```
void main() {
    int y = fact(2);
    int z = fact(3);
    int w = fact(getInputFromUser());
}
```

We can analyze the first two calls to `fact` within `main` straightforwardly, and in fact we can even cache the results of analyzing `fact(2)` for reuse when analyzing the recursive call inside `fact(3)`.

For the third call to `fact`, the argument is determined at runtime, and so constant propagation uses $\top$ for the calling context. In this case, the recursive call to `fact()` also has $\top$ as the calling context. But we cannot look up the result in the cache yet as analysis of `fact()` with $\top$ has not completed. A naive approach would attempt to analyze `fact()` with $\top$ again, and would therefore not terminate.

We can solve the problem by applying the same idea as in intraprocedural analysis. The recursive call is a kind of a loop. We make the initial assumption that the result of the recursive call is $\bot$, conceptually equivalent to information coming from the back edge of a loop. When we discover the result is a higher point in the lattice then $\bot$, we reanalyze the calling context (and recursively, all calling contexts that depend on it). The algorithm to do so can be expressed as follows:

type $Context$
  val $fn : Function$ ▷ the function being called
  val $input : \sigma$ ▷ input for this set of calls

type $Summary$ ▷ the input/output summary for a context
  val $input : \sigma$
  val $output : \sigma$

val $worklist : Set[Context]$ ▷ contexts we must revisit due to updated analysis information
val $analyzing : Set[Context]$ ▷ the contexts we are currently analyzing
val $results : Map[Context, Summary]$ ▷ the analysis results
val $callers : Map[Context, Set[Context]]$ ▷ the call graph - used for change propagation

function GETCTX($f, callingCtx, n, \sigma_{in}$)
  return $Context(f, \sigma_{in})$ ▷ constructs a new $Context$ with f and σ_{in}
end function

function ANALYZEPROGRAM ▷ starting point for interprocedural analysis
  $initCtx \leftarrow$ GETCTX(`main`, **nil**, $0, \top$)
  $worklist \leftarrow \{initCtx\}$
  $results[initCtx] \leftarrow Summary(\top, \bot)$
  while NOTEMPTY($worklist$) **do**
    $ctx \leftarrow$ REMOVE($worklist$)
    ANALYZE($ctx, results[ctx].input$)
  end while
end function

function ANALYZE(ctx, σ_{in})
  $\sigma_{out} \leftarrow results[ctx].output$
  ADD($analyzing, ctx$)
  $\sigma'_{out} \leftarrow$INTRAPROCEDURAL(ctx, σ_{in})
  REMOVE($analyzing, ctx$)
  if $\sigma'_{out} \not\sqsubseteq \sigma_{out}$ **then**
    $results[ctx] \leftarrow Summary(\sigma_{in}, \sigma_{out} \sqcup \sigma'_{out})$
    for $c \in callers[ctx]$ **do**
      ADD($worklist, c$)
    end for
  end if
  return σ'_{out}
end function

function FLOW($[\![n\colon x := f(y)]\!], ctx, \sigma_n$) ▷ called by intraprocedural analysis
  $\sigma_{in} \leftarrow [formal(f) \mapsto \sigma_n(y)]$ ▷ map f's formal parameter to info on actual from σ_n
  $calleeCtx \leftarrow$ GETCTX(f, ctx, n, σ_{in})
  $\sigma_{out} \leftarrow$RESULTSFOR$(calleeCtx, \sigma_{in})$
  ADD$(callers[calleeCtx], ctx)$
  return $\sigma_n[x \mapsto \sigma_{out}[result]]$ ▷ update dataflow with the function's result
end function

function RESULTSFOR(ctx, σ_{in})
  if $ctx \in \text{dom}(results)$ **then**
    if $\sigma_{in} \sqsubseteq results[ctx].input$ **then**
      return $results[ctx].output$ ▷ existing results are good
    else
      $results[ctx].input \leftarrow results[ctx].input \sqcup \sigma_{in}$ ▷ keep track of more general input
    end if
  else
    $results[ctx] = Summary(\sigma_{in}, \bot)$ ▷ initially optimistic assumption
  end if
  if $ctx \in analyzing$ **then**
    return $results[ctx].output$ ▷ $\bot$ if it hasn't been analyzed yet; otherwise last known
  else
    return ANALYZE$(ctx, results[ctx].input)$
  end if
end function

The following example shows that the algorithm generalizes naturally to the case of mutually recursive functions:

```
bar() { if (...) return 2 else return foo() }
foo() { if (...) return 1 else return bar() }

main() { foo(); }
```

8.4 Precision and Termination

Precision. A notable part of the algorithm above is that if we are currently analyzing a context and are asked to analyze it again, we return $\bot$ as the result of the analysis. This has similar benefits to using $\bot$ for initial dataflow values on the back edges of loops: starting with the most optimistic assumptions about code we haven't finished analyzing allows us to reach the best possible fixed point. The following example program illustrates a function where the result of analysis will be better if we assume $\bot$ for recursive calls to the same context, vs. for example if we assumed $\top$:

```
int iterativeIdentity(x, y)
    if x <= 0
        return y
    else
        return iterativeIdentity(x-1, y)

void main(z)
    w = iterativeIdentity(z, 5)
```

Termination. When will the algorithm above terminate? *Analyze* is called only when (1) a context has not been analyzed yet, or when (2) it has just been taken off the worklist. So it is

called once per reachable context, plus once for every time a reachable context is added to the worklist.

We can bound the total number of worklist additions by (C) the number of reachable contexts, times (H) the height of the lattice (we don't add to the worklist unless results for some context changed, i.e. went up in the lattice relative to an initial assumption of $\perp$ or relative to the last analysis result), times (N) the number of callers of that reachable context. C*N is just the number of edges (E) in the inter-context call graph, so we can see that we will do intraprocedural analysis O(E*H) times.

Thus the algorithm will terminate as long as the lattice is of finite height and there are a finite number of reachable contexts. Note, however, that for some lattices, notably including constant propagation, there are an unbounded number of lattice elements and thus an unbounded number of contexts. If more than a finite number are not reachable, the algorithm will not terminate. So for lattices with an unbounded number of elements, we need to adjust the context-sensitivity approach above to limit the number of contexts that are analyzed.

8.5 Approaches to Limiting Context-Sensitivity

No context-sensitivity. One approach to limiting the number of contexts is to allow only one for each function. This is equivalent to the interprocedural control flow graph approach described above. We can recast this approach as a variant of the generic interprocedural analysis algorithm by replacing the *Context* type to track only the function being called, and then having the GETCTX method always return the same context:

type $Context$
  val $fn : Function$

function GETCTX($f, callingCtx, n, \sigma_{in}$)
  return $Context(f)$
end function

Note that in this approach the same calling context might be used for several different input dataflow information σ_{in}, one for each call to GETCTX. This is handled correctly by RESULTSFOR, which updates the input information in the *Summary* for that context so that it generalizes all the input to the function seen so far.

Limited contexts. Another approach is to create contexts as in the original algorithm, but once a certain number of contexts have been created for a given function, merge all subsequent calls into a single context. Of course, this means the algorithm will lose precision beyond this bounds. But, if most functions have fewer contexts that are actually used, this can be a good strategy for analyzing most of the program in a context-sensitive way while avoiding performance problems for the minority of functions that are called from many different contexts.
Can you implement a GETCTX *function that represents this strategy?*

Call strings. Another context sensitivity strategy is to differentiate contexts by a *call string*: the call site, its call site, and so forth. For example, the initial context for the `main` function is $\langle$`main`$, [] \rangle$; a call to `foo` on line 3 will create the context $\langle$`foo`$, [3] \rangle$; if this function calls `bar` on line 10 we will get the context $\langle$`bar`$, [3, 10] \rangle$, and so on. If `main` additionally calls `foo` on line 5, we will get contexts $\langle$`foo`$, [5] \rangle$ and then eventually $\langle$`bar`$, [5, 10] \rangle$. So, the two calls to `bar` are distinguished by the full calling context from `main`. In the limit, when considering call strings of arbitrary length, this provides full context sensitivity (but is not guaranteed to terminate for arbitrary recursive functions). This strategy can be implemented using the following representation in pseudo-code:

type $Context$
  val $fn : Function$
  val $string : List[Int]$

function GETCTX($f, callingCtx, n, \sigma_{in}$)
  $newStr \leftarrow callingCtx.string$ ++ n
  return $Context(f, newStr)$
end function

Dataflow analysis results for contexts based on arbitrarylength call strings are as precise as the results for contexts based on separate analysis for each different input dataflow information. The latter strategy can be more efficient, however, because it reuses analysis results when a function is called twice with different call strings but the same input dataflow information.

In practice, both strategies (arbitrary-length call strings vs. input dataflow information) can result in reanalyzing each function an unacceptable number of times. Multiple contexts must be combined somehow. The call-string approach provides an easy, but naive, way to do this: call strings can simply be cut off at a certain length. For example, if we have call strings "a b c" and "d e b c" (where c is the most recent call site) with a cutoff of 2, the input dataflow information for these two call strings will be merged and the analysis will be run only once, for the context identified by the common length-two suffix of the strings, "b c". We can illustrate this by redoing the analysis of the factorial example. The algorithm is the same as above; however, we use a different implementation of GETCTX that computes the call string suffix:

type $Context$
  val $fn : Function$
  val $string : List[Int]$

function GETCTX($f, callingCtx, n, \sigma_{in}$)
  $newStr \leftarrow$SUFFIX($callingCtx.string$ ++ n, CALL_STRING_CUTOFF)
  return $Context(f, newStr)$
end function

Although this strategy reduces the overall number of analyses, it does so in a relatively blind way. If a function is called many times but we only want to analyze it a few times, we want to group the calls into analysis contexts so that their input information is similar. Call string context is a heuristic way of doing this that sometimes works well. But it can be wasteful: if two different call strings of a given length happen to have exactly the same input analysis information, we will do an unnecessary extra analysis, whereas it would have been better to spend that extra analysis to differentiate calls with longer call strings that have different analysis information.

Given a limited analysis budget, it is usually best to use heuristics that are directly based on input information. Unfortunately these heuristics are harder to design, but they have the potential to do much better than a call-string based approach. We will look at some examples from the literature to illustrate this later in the course.

Chapter 9

Control Flow Analysis for Functional Languages

We have made progress by expanding our dataflow analysis to handle programs with multiple procedures. However, the approach we've developed relies on a number of simplifying assumptions. Notably, in WHILE3ADDR with functions, it is always easy to tell which function is being called at any particular callsite. This is often *not* the case in real languages. Object-oriented languages (or any language with dynamic dispatch) and functional languages challenge this assumption: in both cases, it can be difficult to tell which function is being called, statically.

We therefore turn now to the general problem of statically analyzing functional languages. In doing so, we will see techniques for addressing this general question of determining control flow (or call graphs), and generalize several of our ideas about dataflow analysis (like the idea of a program point). Additionally, analyzing functional languages motivates and provides a good introduction to *constraint-based analyses*. We will additionally expand on a number of these ideas in subsequent classes.

9.1 A simple, labeled, functional language

Consider an idealized functional language based on the lambda calculus, similar to the core of Scheme or ML, with the additional property that we *label* all expressions:

e	$\in$	$Expressions$	...or labelled terms
t	$\in$	$Term$	...or unlabelled expressions
l	$\in$	$\mathcal{L}$	labels

e	::=	t^l
t	::=	$\lambda x.e$
	|	x
	|	$(e_1)\ (e_2)$
	|	let $x = e_1$ in e_2
	|	if e_0 then e_1 else e_2
	|	$n \mid\ e_1 + e_2 \mid$ ...

The grammar includes a definition of an anonymous function $\lambda x.e$, where x is the function argument and e is the function body.[1] The function can include any of the other types of ex-

[1]The formulation in PPA also includes a syntactic construct for explicitly recursive functions. The ideas extend naturally, but we'll follow the simpler syntax for expository purposes.

pressions, such as variables x or function calls $(e_1^a)(e_2^b)$,[2] where e_1 is the function to be invoked and e_2 is passed to that function as an argument (labeled a and b respectively). We evaluate a function call $(\lambda x.e)(v)$ by substituting the argument v for all occurrences of x in e. For example, $((\lambda x.(x^a + 1^b)^c)^d(3)^e)^g$ evaluates to $3+1$, which of course evaluates to 4. A more interesting example is $(((\lambda f.(f^a\ 3^b)^c)^e(\lambda x.(x^g + 1^h)^i)^j)^k$, which first substitutes the argument for f, yielding $(\lambda x.x^g + 1^h)^i\ 3$. Then we invoke the function, getting $3 + 1$ which again evaluates to 4.

Note that this grammar associates each expression with a label $l \in \mathcal{L}$; this is important to keeping track of analysis information (analogous to program points in our imperative analysis), as we discuss next.

9.2 Simple Control Flow Analysis

Static analysis can be just as useful in this type of language as in imperative languages, but immediate complexities arise. For example: what is a *program point* in a language without obvious predecessors or successors? Computation is intrinsically nested. Second, because functions are first-class entities that can be passed around as variables, it's not obvious which function is being applied where. We need some way to figure this out, because the value a function returns (which we may hope to track, such as through constant propagation analysis) will inevitably depend on which function is called, as well as its arguments.

Control flow analysis (CFA)[3] seeks to statically determine which functions could be associated with which variables. Because functional languages are not based on statements but rather expressions, it is appropriate to reason about both the values of variables and the values expressions evaluate to.

9.2.1 0-CFA

We will start by discussing the simplest form of a CFA, called 0-CFA. This is the simplest form because it is context-insensitive (the "0-" label indicates no context is taken into account). We track analysis information for variables and labels, in lieu of the explicit program points in the control flow graphs we used before. Although this may feel like a big change, this approach actually connects directly to what we've been doing in imperative dataflow analysis so far. Dataflow analysis is a type of *abstract interpretation*, an overall framework or theory of sound approximation of program semantics. At a high level and separate from a particular program definition, abstract interpretation associates *labels* with *properties* by manipulating sets of states using monotonic functions over ordered sets as defined by lattices. In our formulation for imperative languages, we implicitly associated labels with the program points between nodes in a control flow graph.

That said, our analysis information σ maps each variable and label to a lattice value. 0-CFA analysis is only concerned with tracking which functions are possibly associated with each location or variable (we will add dataflow information later), and so the abstract domain is as follows:

$$\sigma \in Var \cup \mathcal{L} \to L \qquad L = \top + \mathcal{P}(\lambda x.e)$$

The analysis information at any given expression is the set of all functions that could be the result of evaluating that expression. As suggested above, expressions are identified by their labels l, and we track similar information for variables. We use $\top$ to denote all possible

[2] In an imperative language this would more typically be written $e_1^a(e_2)^b$, but we follow the functional convention here, with parenthesis included when helpful syntactically.

[3] This nomenclature is confusing because it is also used to refer to analyses of control flow graphs in imperative languages; We usually abbreviate to CFA when discussing the analysis of functional languages.

functions; if we know all the functions in the program, we could enumerate them, but a symbolic $\top$ representation is useful when we don't have the whole program available.

Question: what is the $\sqsubseteq$ relation on this dataflow state?

A 0-CFA is a *Constraint Based Analysis*: it is defined via inference rules that generate constraints over the possible dataflow values for each variable or labeled location; those constraints are then solved. We use the $\hookrightarrow$ to define constraint generation. The judgment $[\![e]\!]^l \hookrightarrow C$ can be read as "The analysis of expression e with label l generates constraints C over dataflow state σ." For our first CFA, we can define inference rules for this judgment as follows:

$$\frac{}{[\![x]\!]^l \hookrightarrow \sigma(x) \sqsubseteq \sigma(l)} \textit{ var}$$

In this rule, the variable value flows to the program location l. Although we didn't list it above (we generalize it below), a rule for constants produces the empty set, because this analysis is tracking only function values.

The rules for functions/calls is more complex:

$$\frac{[\![e]\!]^{l_0} \hookrightarrow C}{[\![\lambda x.e^{l_0}]\!]^l \hookrightarrow \{\lambda x.e\} \sqsubseteq \sigma(l) \cup C} \textit{ lambda}$$

$$\frac{[\![e_1]\!]^{l_1} \hookrightarrow C_1 \quad [\![e_2]\!]^{l_2} \hookrightarrow C_2}{[\![e_1^{l_1}\ e_2^{l_2}]\!]^l \hookrightarrow C_1 \cup C_2 \cup \textbf{fn}\ l_1 : l_2 \Rightarrow l} \textit{ apply}$$

The first rule just states that if a literal function is declared at a program location l, that function is part of the lattice value $\sigma(l)$ computed by the analysis for that location. Because we want to analyze the data flow inside the function, we also generate a set of constraints C from the function body and return those constraints as well.

The rule for application first analyzes the function and the argument to extract two sets of constraints C_1 and C_2. We then generate an abstract *function flow constraint* of the form $\textbf{fn}\ l_1 : l_2 \Rightarrow l$. This function flow constraint is interpreted by the constraint solver to generate additional concrete constraints using the following rule:

$$\frac{\lambda x.e_0^{l_0} \in \sigma(l_1)}{\textbf{fn}\ l_1 : l_2 \Rightarrow l \hookrightarrow \sigma(l_2) \sqsubseteq \sigma(x) \wedge \sigma(l_0) \sqsubseteq \sigma(l)} \textit{ function-flow}$$

This rule states that for every literal function $\lambda x.e_0^{l_0}$ that the analysis (eventually) determines the expression labeled l_1 may evaluate to, we must generate additional constraints that capture value flow from the actual argument expression l_2 to formal function argument x, and from the function result to the calling expression l.

Consider the first example program given above: $((\lambda x.(x^a + 1^b)^c)^d(3)^e)^g$. The first rule to use is *apply* (because that's the top-level program construct). We will work this out together, but the generated constraints could look like:

$$(\sigma(x) \sqsubseteq \sigma(a)) \cup (\{\lambda x.x + 1\} \sqsubseteq \sigma(d)) \cup (\sigma(e) \sqsubseteq \sigma(x)) \wedge (\sigma(c) \sqsubseteq \sigma(g))$$

There are many possible valid (typically referred to as *acceptable*) solutions to this constraint set. Eliding the formalities, it suffices to say that we would like the least solution to these constraints, as that will be the most precise result. We will return to constraint solving properly later in the course; for now, we will simply assert that a σ that maps all variables and locations except d to $\varnothing$, and d to $\{\lambda x.x + 1\}$, satisfies this set of constraints.

Question: what might the rules for the if-then-else or arithmetic operator expressions look like?

9.2.2 0-CFA with dataflow information

The analysis in the previous subsection is interesting if all you're interested in is which functions can be called where, but doesn't solve the general problem of dataflow analysis of functional programs. Fortunately, extending that approach to a more general analysis space is straightforward: we simply add the abstract information we're tracking to the abstract domain defined above. For constant propagation, for example, we can extend the dataflow state as follows:

$$\sigma \in \mathit{Var} \cup \mathit{Lab} \to L \qquad L = \mathbb{Z} + \top + \mathcal{P}(\lambda x.e)$$

Now, the analysis information maps each program point (or variable) to an integer n, or $\top$, or a set of functions. This requires that we modify our inference rules slightly, but not as much as you might expect. Indeed, the rules mostly change for arithmetic operators (which we omitted above) and constants. We simply need to provide an abstraction over concrete values that captures the dataflow information in question. We get the following rules:

$$\frac{}{[\![n]\!]^l \hookrightarrow \alpha(n) \sqsubseteq \sigma(l)}\ \mathit{const} \qquad \frac{[\![e_1]\!]^{l_1} \hookrightarrow C_1 \quad [\![e_2]\!]^{l_2} \hookrightarrow C_2}{[\![e_1^{l_1} + e_2^{l_2}]\!]^l \hookrightarrow C_1 \cup C_2 \cup (\sigma(l_1) +_\top \sigma(l_2)) \sqsubseteq \sigma(l)}\ \mathit{plus}$$

Where α is defined as we discussed in abstract interpretation, and $+_\top$ is addition lifted to work over a domain that includes $\top$ (and simply ignores/drops any lambda values). There are similar rules for other arithmetic operations.

Consider the second example, above, properly labeled: $(((\lambda f.(f^a\ 3^b)^c)^e(\lambda x.(x^g + 1^h)^i)^j)^k$ A constant propagation analysis could produce the following results:

$\mathit{Var} \cup \mathit{Lab}$	L	by rule
e	$\lambda f.f\ 3$	lambda
j	$\lambda x.x + 1$	lambda
f	$\lambda x.x + 1$	apply
a	$\lambda x.x + 1$	var
b	3	const
x	3	apply
g	3	var
h	1	const
i	4	add
c	4	apply
k	4	apply

9.3 m-Calling Context Sensitive Control Flow Analysis (m-CFA)

The control flow analysis described above quickly becomes imprecise in more interesting programs that reuse functions in several calling contexts. This problem should seem familiar from interprocedural imperative program analysis, but the following code illustrates the problem in this new language:

let *add* $= \lambda x.\ \lambda y.\ x + y$
let *add5* $= (add\ 5)^{a5}$
let *add6* $= (add\ 6)^{a6}$
let *main* $= (add5\ 2)^m$

This example illustrates *currying*, in which a function such as *add* that takes two arguments x and y in sequence can be called with only one argument (e.g. 5 in the call labeled $a5$), resulting in a function that can later be called with the second argument (in this case, 2 at the call labeled m). The value 5 for the first argument in this example is stored with the function in the *closure add5*. Thus when the second argument is passed to *add5*, the closure holds the value of x so that the sum $x + y = 5 + 2 = 7$ can be computed.

In this case, we create two closures, *add5* and *add6*, binding 5 and 6 and the respective values for x. 0-CFA analysis cannot distinguish them, and because it only computes one value for x we learn only that x has the value $\top$. This is illustrated in the following analysis (we shorten the trace to focus only on the variables):

Var $\cup$ *Lab*	L	notes
add	$\lambda x.\ \lambda y.\ x + y$	
x	5	when analyzing first call
$add5$	$\lambda y.\ x + y$	
x	$\top$	when analyzing second call
$add6$	$\lambda y.\ x + y$	
$main$	$\top$	

We can add precision using a context-sensitive analysis. One could, in principle, use either the functional or call-string approach we discussed previously. In practice the call-string approach is more commonly used for control-flow analysis in functional programming languages, perhaps because functional programs will typically produced an intractable number of contexts per function, and it is easier to place a bound on the analysis in the call-string approach.

We add context sensitivity by making our analysis information σ track information separately for different call strings, denoted by Δ. Here a call string is a sequence of labels, each one denoting a function call site, where the sequence can be of any length between 0 and some bound m (in practice m will be in the range 0-2 for scalability reasons):

$$\sigma \in (\mathit{Var} \cup \mathit{Lab}) \times \Delta \to L \qquad \Delta = \mathit{Lab}^{n \leqslant m} \qquad L = \mathbb{Z} + \top + \mathcal{P}((\lambda x.e, \delta))$$

When a lambda expression is analyzed, we now consider as part of the lattice the call string context δ in which its free variables were captured. We can then define a set of rules that generate constraints which, when solved, provide an answer to control-flow analysis, as well as (in this case) constant propagation:

$$\frac{}{\delta \vdash [\![n]\!]^l \hookrightarrow \alpha(n) \sqsubseteq \sigma(l, \delta)}\ \textit{const} \qquad \frac{}{\delta \vdash [\![x]\!]^l \hookrightarrow \sigma(x, \delta) \sqsubseteq \sigma(l, \delta)}\ \textit{var}$$

$$\frac{}{\delta \vdash [\![\lambda x.e^{l_0}]\!]^l \hookrightarrow \{(\lambda x.e, \delta)\} \sqsubseteq \sigma(l, \delta)}\ \textit{lambda}$$

$$\frac{\delta \vdash [\![e_1]\!]^{l_1} \hookrightarrow C_1 \qquad \delta \vdash [\![e_2]\!]^{l_2} \hookrightarrow C_2}{\delta \vdash [\![e_1^{l_1}\ e_2^{l_2}]\!]^l \hookrightarrow C_1 \cup C_2 \cup \mathbf{fn}_\delta\ l_1 : l_2 \Rightarrow l}\ \textit{apply}$$

These rules contain a call string context δ in which the analysis of each line of code is done. The rules *const* and *var* are unchanged except for indexing σ by the current context δ. Similarly, the *apply* rule is the same except we index everything by δ and record δ as part of the function flow constraint. The *lambda* rule now captures the context δ along with the lambda expression, so that when the lambda expression is called the analysis knows in which context to look up the free variables. But the rule no longer analyzes inside the function; we want to delay that and do it for a new context δ' when the function is called.

$$\frac{\begin{array}{c}(\lambda x.e_0^{l_0}, \delta) \in \sigma(l_1, \delta) \qquad \delta' = \mathit{suffix}(\delta{+}{+}l, m) \\ C_1 = \sigma(l_2, \delta) \sqsubseteq \sigma(x, \delta') \wedge \sigma(l_0, \delta') \sqsubseteq \sigma(l, \delta) \\ C_2 = \{\sigma(y, \delta) \sqsubseteq \sigma(y, \delta') \mid y \in FV(\lambda x.e_0)\} \\ \delta' \vdash [\![e_0]\!]^{l_0} \hookrightarrow C_3\end{array}}{\mathbf{fn}_\delta\ l_1 : l_2 \Rightarrow l \hookrightarrow C_1 \cup C_2 \cup C_3}\ \textit{function-flow-}\delta$$

The function flow constraint has gotten a bit more complicated. A new context δ' is formed by appending the current call site l to the old call string, then taking the suffix of length m (or less). For each function that may be called, we set up constraints between the actual and formal parameters and the function result, as before (C_1). We analyze the body of the function in the new context δ' (C_3). Finally, we produce constraints that bind the free variables in the new context: all free variables in the called function flow from the point δ_0 at which the closure was captured.

We can now reanalyze the earlier example, observing the benefit of context sensitivity. In the table below, • denotes the empty calling context (e.g. when analyzing the *main* procedure):

Var / Lab, δ	L	notes
add, •	$(\lambda x.\ \lambda y.\ x + y,\ \bullet)$	
x, a5	5	
add5, •	$(\lambda y.\ x + y,\ a5)$	
x, a6	6	
add6, •	$(\lambda y.\ x + y,\ a6)$	
main, •	7	

Note three points about this analysis. First, we can distinguish the values of x in the two calling contexts: x is 5 in the context a5 but it is 6 in the context a6. Second, the closures returned to the variables *add5* and *add6* record the scope in which the free variable x was bound when the closure was captured. This means, third, that when we invoke the closure *add5* at program point m, we will know that x was captured in calling context a5, and so when the analysis analyzes the addition, it knows that x holds the constant 5 in this context. This enables constant propagation to compute a precise answer, learning that the variable *main* holds the value 7.

Optional: Uniform k-Calling Context Sensitive Control Flow Analysis (k-CFA)

m-CFA was proposed recently by Might, Smaragdakis, and Van Horn as a more scalable version of the original k-CFA analysis developed by Shivers for Scheme. While m-CFA now seems to be a better tradeoff between scalability and precision, k-CFA is interesting both for historical reasons and because it illustrates a more precise approach to tracking the values of variables in a closure. The following example illustrates a situation in which m-CFA may be too imprecise:

$$\begin{array}{ll}\textbf{let}\ \mathit{adde} & = \lambda x. \\ & \quad \textbf{let}\ h = \lambda y.\ \lambda z.\ x + y + z \\ & \quad \textbf{let}\ r = h\ 8 \\ & \quad \textbf{in}\ r \\ \textbf{let}\ t & = (\mathit{adde}\ 2)^t \\ \textbf{let}\ f & = (\mathit{adde}\ 4)^f \\ \textbf{let}\ e & = (t\ 1)^e\end{array}$$

When we analyze it with m-CFA, we get the following results:

Var / Lab, δ	L	notes
adde, $\bullet$	$(\lambda x..., \bullet)$	
x, t	2	
y, r	8	
x, r	2	when analyzing first call
t, $\bullet$	$(\lambda z.\ x + y + z,\ r)$	
x, f	4	
x, r	$\top$	when analyzing second call
f, $\bullet$	$(\lambda z.\ x + y + z,\ r)$	
t, $\bullet$	$\top$	

The k-CFA analysis is like m-CFA, except that rather than keeping track of the scope in which a closure was captured, the analysis keeps track of the scope in which each variable captured in the closure was defined. We use an environment η to track this. Note that since η can represent a separate calling context for each variable, it has the potential to be more accurate, but also much more expensive. We can represent the analysis information as follows:

$$\begin{array}{rclcrcl} \sigma & \in & (\mathit{Var} \cup \mathit{Lab}) \times \Delta \to L & & \Delta & = & \mathit{Lab}^{n \leqslant k} \\ L & = & \mathbb{Z} + \top + \mathcal{P}(\lambda x.e, \eta) & & \eta & \in & \mathit{Var} \to \Delta \end{array}$$

Let us briefly analyze the complexity of this analysis. In the worst case, if a closure captures n different variables, we may have a different call string for each of them. There are $O(n^k)$ different call strings for a program of size n, so if we keep track of one for each of n variables, we have $O(n^{n*k})$ different representations of the contexts for the variables captured in each closure. This exponential blowup is why k-CFA scales so badly. m-CFA is comparatively cheap—there are "only" $O(n^k)$ different contexts for the variables captured in each closure—still exponential in k, but polynomial in n for a fixed (and generally small) k.

We can now define the rules for k-CFA. They are similar to the rules for m-CFA, except that we now have two contexts: the calling context δ, and the environment context η tracking the context in which each variable is bound. When we analyze a variable x, we look it up not in the current context δ, but the context $\eta(x)$ in which it was bound. When a lambda is analyzed, we track the current environment η with the lambda, as this is the information necessary to determine where captured variables are bound. The function flow rule is actually somewhat simpler, because we do not copy bound variables into the context of the called procedure:

$$\frac{}{\delta, \eta \vdash [\![n]\!]^l \hookrightarrow \alpha(n) \sqsubseteq \sigma(l, \delta)}\ \textit{const} \qquad \frac{}{\delta, \eta \vdash [\![x]\!]^l \hookrightarrow \sigma(x, \eta(x)) \sqsubseteq \sigma(l, \delta)}\ \textit{var}$$

$$\frac{}{\delta, \eta \vdash [\![\lambda x.e^{l_0}]\!]^l \hookrightarrow \{(\lambda x.e, \eta)\} \sqsubseteq \sigma(l, \delta)}\ \textit{lambda}$$

$$\frac{\delta, \eta \vdash [\![e_1]\!]^{l_1} \hookrightarrow C_1 \qquad \delta, \eta \vdash [\![e_2]\!]^{l_2} \hookrightarrow C_2}{\delta, \eta \vdash [\![e_1^{l_1}\ e_2^{l_2}]\!]^l \hookrightarrow C_1 \cup C_2 \cup \mathbf{fn}_\delta\ l_1 : l_2 \Rightarrow l}\ \textit{apply}$$

$$\frac{\begin{array}{c}(\lambda x.e_0^{l_0}, \eta_0) \in \sigma(l_1) \qquad \delta' = \mathit{suffix}(\delta{+}{+}l, m) \\ C_1 = \sigma(l_2, \delta) \sqsubseteq \sigma(x, \delta') \wedge \sigma(l_0, \delta') \sqsubseteq \sigma(l, \delta) \\ \delta', \eta_0 \vdash [\![e_0]\!]^{l_0} \hookrightarrow C_2 \end{array}}{\mathbf{fn}_\delta\ l_1 : l_2 \Rightarrow l \hookrightarrow C_1 \cup C_2}\ \textit{function-flow-}\delta$$

Now we can see how k-CFA analysis can more precisely analyze the latest example program. In the simulation below, we give two tables: one showing the order in which the functions are analyzed, along with the calling context δ and the environment η for each analysis, and the other as usual showing the analysis information computed for the variables in the program:

function	δ	η
main	$\bullet$	$\varnothing$
adde	t	$\{x \mapsto t\}$
h	r	$\{x \mapsto t,\ y \mapsto r\}$
adde	f	$\{x \mapsto f\}$
h	r	$\{x \mapsto f,\ y \mapsto r\}$
$\lambda z....$	e	$\{x \mapsto t,\ y \mapsto r,\ z \mapsto e\}$

Var / Lab, δ	L	notes
adde, $\bullet$	$(\lambda x...,\ \bullet)$	
x, t	2	
y, r	8	
t, $\bullet$	$(\lambda z.\ x + y + z,\ \{x \mapsto t,\ y \mapsto r\})$	
x, f	4	
f, $\bullet$	$(\lambda z.\ x + y + z,\ \{x \mapsto f,\ y \mapsto r\})$	
z, e	1	
t, $\bullet$	11	

Tracking the definition point of each variable separately is enough to restore precision in this program. However, programs with this structure—in which analysis of the program depends on different calling contexts for bound variables even when the context is the same for the function eventually called—appear to be rare in practice. Might et al. observed no examples among the real programs they tested in which k-CFA was more accurate than m-CFA—but k-CFA was often far more costly. Thus at this point the m-CFA analysis seems to be a better tradeoff between efficiency and precision, compared to k-CFA.

Chapter 10

Advanced Interprocedural Analysis: Pointer Analysis and Object-Oriented Call Graph Construction

We have successfully extended our interprocedural dataflow analysis framework to a small functional programming language, which required us to reason explicitly about which functions might be called, where. This provides insight into similar problems in other programming paradigms, namely *dynamic dispatch*. Precisely addressing dynamic dispatch relies on techniques for *pointer analysis*, which establishes which pointers can point to which locations. Analyses that address real programming languages (whether they use dynamic dispatch or not) must address pointers, because ignoring them dramatically impacts analysis precision. Thus, in the interest of adapting our framework to real languages, we turn our attention to these issues.

10.1 Pointer Analysis

Pointers are variables whose value refers to another value elsewhere in memory, by storing the address of that stored value. To illustrate why they matter in analyzing real programs, consider constant-propagation analysis of the following program:

$$\begin{array}{ll} 1: & z := 1 \\ 2: & p := \&z \\ 3: & *p := 2 \\ 4: & \text{print } z \end{array}$$

To analyze this program correctly we must be aware that at instruction 3, p points to z. If this information is available we can use it in a flow function as follows:

$$f_{CP}[\![*p := y]\!](\sigma) \quad = \sigma[z \mapsto \sigma(y) \mid z \in \textit{must-point-to}(p)]$$

When we know exactly what a variable x points to, we have *must-point-to* information, and we can perform a *strong update* of the target variable z, because we know with confidence that assigning to $*p$ assigns to z. A technicality in the rule is quantifying over all z such that p must point to z. How is this possible? It is not possible in C or Java; however, in a language with pass-by-reference, for example C++, it is possible that two names for the same location are in scope.

Of course, it is also possible to be uncertain to which of several distinct locations p points:

$$
\begin{array}{ll}
1: & z := 1 \\
2: & \textbf{if } (\mathit{cond})\ p := \&y \textbf{ else } p := \&z \\
3: & *p := 2 \\
4: & \textbf{print } z
\end{array}
$$

Now constant propagation analysis must conservatively assume that z could hold either 1 or 2. We can represent this with a flow function that uses may-point-to information:

$$f_{CP}[\![*p := y]\!](\sigma) \quad = \sigma[z \mapsto \sigma(z) \sqcup \sigma(y) \mid z \in \mathit{may\text{-}point\text{-}to}(p)]$$

10.1.1 Andersen's Points-To Analysis

Two common kinds of pointer analysis are *alias analysis* and *points-to* analysis. Alias analysis computes sets S holding pairs of variables (p, q), where p and q may (or must) point to the same location. Points-to analysis computes the set $\mathit{points\text{-}to}(p)$, for each pointer variable p, where the set contains a variable x if p may (or must) point to the location of the variable x. We will focus primarily on points-to analysis, beginning with a simple but useful approach originally proposed by Andersen.[1]

Our initial setting will be C programs. We are interested in analyzing instructions that are relevant to pointers in the program. Ignoring for the moment memory allocation and arrays, we can decompose all pointer operations in C into four types:

$$
\begin{array}{lll}
I ::= & \ldots & \\
\mid & p := \&x & \text{taking the address of a variable} \\
\mid & p := q & \text{copying a pointer from one variable to another} \\
\mid & *p := q & \text{assigning through a pointer} \\
\mid & p := *q & \text{dereferencing a pointer}
\end{array}
$$

Andersen's points-to analysis is a context-insensitive interprocedural analysis. It is also a *flow-insensitive analysis*, that is an analysis that does not consider program statement order. Context- and flow-insensitivity improve analysis performance, as precise pointer analysis can be notoriously expensive.

We will formulate Andersen's analysis by generating set constraints which can later be processed by a set constraint solver, much like we did for CFA. Because the analysis is flow-insensitive, we do not care what order the instructions in the program come in; we simply generate a set of constraints and solve them. Constraint generation for each statement works by these rules:

$$\frac{}{[\![p := \&x]\!] \hookrightarrow l_x \in p}\ \textit{address-of}$$

$$\frac{}{[\![p := q]\!] \hookrightarrow p \supseteq q}\ \textit{copy}$$

$$\frac{}{[\![*p := q]\!] \hookrightarrow *p \supseteq q}\ \textit{assign}$$

$$\frac{}{[\![p := *q]\!] \hookrightarrow p \supseteq *q}\ \textit{dereference}$$

The first rule states that a constant location l_x, representation the address of x, is in the set of location pointed to by p. The second rule states that the set of locations pointed to by

[1] PhD thesis: "Program Analysis and Specialization for the C Programming Language."

p must be a superset of those pointed to by q. The last two rules state the same, but take into account that one or the other pointer is dereferenced. Note that if Andersen's algorithm says that the set p points to only one location l_z, we have *must-point-to* information, whereas if the set p contains more than one location, we have only *may-point-to* information.

A number of specialized set constraint solvers exist, and constraints in the form above can be translated into input for them.[2] We will treat constraint-solving abstractly using the following constraint propagation rules:

$$\frac{p \supseteq q \quad l_x \in q}{l_x \in p} \textit{ copy}$$

$$\frac{*p \supseteq q \quad l_r \in p \quad l_x \in q}{l_x \in r} \textit{ assign}$$

$$\frac{p \supseteq *q \quad l_r \in q \quad l_x \in r}{l_x \in p} \textit{ dereference}$$

We can now apply Andersen's points-to analysis to the programs above. We can also apply it to programs with dynamic memory allocation, such as:

$$\begin{array}{ll} 1: & q := \textit{malloc}() \\ 2: & p := \textit{malloc}() \\ 3: & p := q \\ 4: & r := \&p \\ 5: & s := \textit{malloc}() \\ 6: & *r := s \\ 7: & t := \&s \\ 8: & u := *t \end{array}$$

The analysis is run the same way, but we treat the memory cell allocated at each *malloc* or *new* statement as an abstract location labeled by the location n of the allocation point:

$$\frac{}{[\![n\text{: } p := \textit{malloc}()]\!] \hookrightarrow l_n \in p} \textit{ malloc}$$

We must be careful because a *malloc* statement can be executed more than once, and each time it executes, a new memory cell is allocated. Unless we have some other means of proving that the malloc executes only once, we must assume that if some variable p only points to one abstract malloc'd location l_n, that is still may-alias information (i.e. p points to only one of the many actual cells allocated at the given program location) and not must-alias information.

Efficiency. Analyzing the efficiency of Andersen's algorithm, we can see that all constraints can be generated in a linear $O(n)$ pass over the program. The solution size is $O(n^2)$, because each of the $O(n)$ variables defined in the program could potentially point to $O(n)$ other variables.

We can derive the execution time as follows:[3] There are $O(n)$ flow constraints generated of the form $p \supseteq q$, $*p \supseteq q$, or $p \supseteq *q$. How many times could a constraint propagation rule fire for each flow constraint? For a $p \supseteq q$ constraint, the rule may fire at most $O(n)$ times, because there are at most $O(n)$ premises of the proper form $l_x \in p$. However, a constraint of the form

[2] Note that the dereference operation (the $*$ in $*p \supseteq q$) is not standard, but can be encoded,

[3] David A. McAllester. 1999. On the Complexity Analysis of Static Analyses. In Proceedings of the 6th International Symposium on Static Analysis (SAS '99): 312–329.

$p \supseteq *q$ could cause $O(n^2)$ rule firings, because there are $O(n)$ premises each of the form $l_x \in p$ and $l_r \in q$. With $O(n)$ constraints of the form $p \supseteq *q$ and $O(n^2)$ firings for each, we have $O(n^3)$ constraint firings overall. A similar analysis applies for $*p \supseteq q$ constraints. McAllester's theorem states that the analysis with $O(n^3)$ rule firings can be implemented in $O(n^3)$ time. Thus we have derived that Andersen's algorithm is cubic in the size of the program, in the worst case.

Interestingly, Andersen's algorithm can be executed in $O(n^2)$ time for *k-sparse* programs.[4] The *k-sparse* assumption requires that at most k statements dereference each variable, and that the flow graph is sparse. The publication showing this result also showed that typical Java programs are *k-sparse*, and that Andersen's algorithm scales quadratically in practice.

10.1.2 Field Sensitivity

What happens when we have a pointer to a struct in C, or an object in an object-oriented language? In this case, we would like the pointer analysis to tell us what each field in the struct or object points to. A simple solution is to be *field-insensitive*, treating all fields in a struct as equivalent. Thus if p points to a struct with two fields f and g, and we assign:

$$\begin{aligned} 1: &\quad p.f := \&x \\ 2: &\quad p.g := \&y \end{aligned}$$

A field-insensitive analysis would tell us (imprecisely) that $p.f$ could point to y. We can modify the rules above by treating any field dereference or field assignment to $p.f$ as a pointer dereference $*p$. Essentially, you can think of this as just considering all fields to be named $*$.

To be more precise, we can instead track the contents each field of each abstract location separately. In the discussion below, we assume a Java-like setting, in which all objects are allocated on the heap and where we cannot take the address of a field. A slightly more complicated variant of this scheme works in C-like languages.

We will use the *malloc* and *copy* rules unchanged from above.[5] We drop the *assign* and *dereference* rules, and replace them with:

$$\frac{}{[\![p := q.f]\!] \hookrightarrow p \supseteq q.f} \textit{ field-read}$$

$$\frac{}{[\![p.f := q]\!] \hookrightarrow p.f \supseteq q} \textit{ field-assign}$$

Now assume that objects (e.g. in Java) are represented by abstract locations l. We will have two forms of basic facts. The first is the same as before: $l_n \in p$, where l_n is an object allocated in a **`new`** statement at line n. The second basic fact is $l_n \in l_m.f$, which states that the field f of the object represented by l_m may point to an object represented by l_n.

We can now process field constraints with the following rules:

$$\frac{p \supseteq q.f \quad l_q \in q \quad l_f \in l_q.f}{l_f \in p} \textit{ field-read}$$

$$\frac{p.f \supseteq q \quad l_p \in p \quad l_q \in q}{l_q \in l_p.f} \textit{ field-assign}$$

If we run this analysis on the code above, we find that it can distinguish that $p.f$ points to x and $p.g$ points to y.

[4] Manu Sridharan and Stephen J. Fink. 2009. The Complexity of Andersen's Analysis in Practice. In Proceedings of the 16th International Symposium on Static Analysis (SAS '09): 205–221.

[5] In Java, the **`new`** expression plays the role of `malloc`

10.1.3 Steensgaard's Points-To Analysis

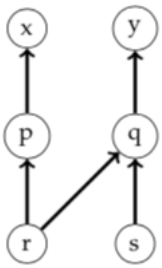
For very large programs, a quadratic-in-practice algorithm is too inefficient. Steensgaard proposed an pointer analysis algorithm that operates in near-linear time, supporting essentially unlimited practical scalability.

The first challenge in designing a near-linear time points-to analysis is to represent the results in linear space. This is nontrivial because over the course of program execution, any given pointer p could potentially point to the location of any other variable or pointer q. Representing all of these pointers explicitly will inherently take $O(n^2)$ space.

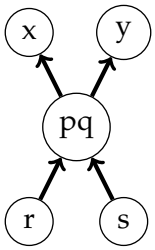
The solution Steensgaard found is based on using constant space for each variable in the program. His analysis associates each variable p with an abstract location named after the variable. Then, it tracks a single points-to relation between that abstract location p and another one q, to which it may point. Now, it is possible that in some real program p may point to both q and some other variable r. In this situation, Steensgaard's algorithm *unifies* the abstract locations for q and r, creating a single abstract location representing both of them. Now we can track the fact that p may point to either variable using a single points-to relationship.

For example, consider the program to the left, and the graph that Andersen's points-to analysis would produce (right):

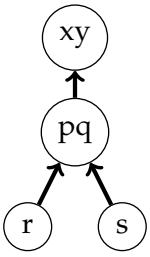
$$
\begin{array}{ll}
1: & p := \&x \\
2: & r := \&p \\
3: & q := \&y \\
4: & s := \&q \\
5: & r := s
\end{array}
$$



But in Steensgaard's setting, when we discover that r could point both to q and to p, we must merge q and p into a single node:



Notice that we have lost precision: by merging the nodes for p and q our graph now implies that s could point to p, which is not the case in the actual program. But we are not done. Now pq has two outgoing arrows, so we must merge nodes x and y. The final graph produced by Steensgaard's algorithm is therefore:



We study Steensgaard's analysis more precisely by specifying a simplified version that ignores function pointers:

$$\frac{}{[\![p := q]\!] \hookrightarrow \mathit{join}(*p, *q)}\ \mathit{copy}$$

$$\frac{}{[\![p := \&x]\!] \hookrightarrow \mathit{join}(*p, x)}\ \mathit{address\text{-}of}$$

$$\frac{}{[\![p := *q]\!] \hookrightarrow \mathit{join}(*p, **q)}\ \mathit{dereference}$$

$$\frac{}{[\![*p := q]\!] \hookrightarrow \mathit{join}(**p, *q)}\ \mathit{assign}$$

With each abstract location p, we associate the abstract location that p points to, denoted $*p$. Abstract locations are implemented as a union-find[6] data structure so that we can merge two abstract locations efficiently. In the rules above, we implicitly invoke *find* on an abstract location before calling *join* on it, or before looking up the location it points to.

The *join* operation essentially implements a union operation on the abstract locations. However, since we are tracking what each abstract location points to, we must update this information also. The algorithm to do so is as follows:

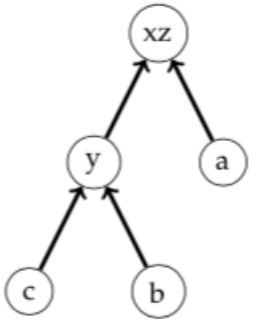
```
join(ℓ1, ℓ2)
    if (find(ℓ1) == find(ℓ2))
        return
    n1 ← *ℓ1
    n2 ← *ℓ2
    union(ℓ1, ℓ2)
    join(n1, n2)
```

Once again, we implicitly invoke *find* on an abstract location before comparing it for equality, looking up the abstract location it points to, or calling *join* recursively.

As an optimization, Steensgaard does not perform the join if the right hand side is not a pointer. For example, if we have an assignment $[\![p := q]\!]$ and q has not been assigned any pointer value so far in the analysis, we ignore the assignment. If later we find that q may hold a pointer, we must revisit the assignment to get a sound result.

Steensgaard illustrated his algorithm using the following program, and the graph the algorithm produces:

```
1 :  a := &x
2 :  b := &y
3 :  if p then
4 :      y := &z
5 :  else
6 :      y := &x
7 :  c := &y
```



Efficiency. Rayside illustrates how Andersen must sometimes do more work than Steensgaard:

[6] See any algorithms textbook

$$
\begin{aligned}
1: &\quad q := \&x \\
2: &\quad q := \&y \\
3: &\quad p := q \\
4: &\quad q := \&z
\end{aligned}
$$

After processing the first three statements, Steensgaard's algorithm will have unified variables x and y, with p and q both pointing to the unified node. Andersen's algorithm will have both p and q pointing to both x and y. When the fourth statement is processed, Steensgaard's algorithm does only a constant amount of work, merging z in with the already-merged xy. On the other hand, Andersen's algorithm must not just create a points-to relation from q to z, but must also propagate that relationship to p. It is this additional propagation step that results in the significant performance difference between these algorithms.[7]

Analyzing Steensgaard's pointer analysis for efficiency, we observe that each of n statements in the program is processed once. The processing is linear, except for *find* operations on the union-find data structure (which may take amortized time $O(\alpha(n))$ each) and the *join* operations. We note that in the *join* algorithm, the short-circuit test will fail at most $O(n)$ times—at most once for each variable in the program. Each time the short-circuit fails, two abstract locations are unified, at cost $O(\alpha(n))$. The unification assures the short-circuit will not fail again for one of these two variables. Because we have at most $O(n)$ operations and the amortized cost of each operation is at most $O(\alpha(n))$, the overall running time of the algorithm is near linear: $O(n * \alpha(n))$. Space consumption is linear, as no space is used beyond that used to represent abstract locations for all the variables in the program text.

Based on this asymptotic efficiency, Steensgaard's algorithm was run on a 1 million line program (Microsoft Word) in 1996; this was an order of magnitude greater scalability than other pointer analyses known at the time.

Steensgaard's pointer analysis is field-insensitive; making it field-sensitive would mean that it is no longer linear.

10.2 Dynamic dispatch

Dynamic dispatch is the process of selecting which implementation of a method or function should be called at runtime; it is a defining characteristic object-oriented programming languages and systems, but is not limited to them (e.g., calling through function pointers in C). To construct a precise call graph in such languages, an analysis must determine the type of the receiver object is at each call site. Flow analysis techniques similar to points-to analysis can be used to compute this information, but using an interprocedural flow analysis off the shelf requires a call graph, which is exactly what we are trying to construct. Therefore, object-oriented call graph construction algorithms must simultaneously build a call graph and compute dataflow information describing the types of the objects to which each variable could point.

10.2.1 Simple approaches

Before examining a full-fledged dataflow analysis-based call graph construction algorithm, we will consider two simpler approaches that do not require flow analysis. These approaches have the side benefit of being very efficient, and so are used in settings such as JIT compilers where analysis time is scarce.

The simplest approach, *class hierarchy analysis*, uses the type of a variable, together with the class hierarchy, to determine what types of object the variable could point to. Unsurprisingly,

[7] For fun, try adding a new statement $r := p$ after statement 3. Then z has to be propagated to the points-to sets of both p and r. In general, the number of propagations can be linear in the number of copies and the number of address-of operators, which makes it quadratic overall even for programs in the simple form above.

this is very imprecise, but can be computed very efficiently in $O(n * t)$ time, because it visits n call sites and at each call site traverses a subtree of size t of the class hierarchy.

An improvement to class hierarchy analysis is *rapid type analysis*, which eliminates from the hierarchy classes that are never instantiated. The analysis iteratively builds a set of instantiated types, method names invoked, and concrete methods called. Initially, it assumes that `main` is the only concrete method that is called, and that no objects are instantiated. It then analyzes concrete methods known to be called, one by one. When a method name is invoked, it is added to the list, and all concrete methods with that name defined within (or inherited by) types known to be instantiated are added to the called list. When an object is instantiated, its type is added to the list of instantiated types, and all its concrete methods that have a method name that is invoked are added to the called list. This proceeds iteratively until a fixed point is reached, at which point the analysis knows all of the object types that may actually be created at run time.

Rapid type analysis can be considerably more precise than class hierarchy analysis in programs that use libraries that define many types, only a few of which are used by the program. It remains extremely efficient, because it only needs to traverse the program once (in $O(n)$ time) and then build the call graph by visiting each of n call sites and considering a subtree of size t of the class hierarchy, for a total of $O(n * t)$ time.

10.2.2 0-CFA Style Object-Oriented Call Graph Construction

Object-oriented call graphs can also be constructed using a pointer analysis such as Andersen's algorithm, either context-insensitive or context-sensitive. The context-sensitive versions are called k-CFA by analogy with control-flow analysis for functional programs. The context-insensitive version is called 0-CFA for the same reason. Essentially, the analysis proceeds as in Andersen's algorithm, but the call graph is built up incrementally as the analysis discovers the types of the objects to which each variable in the program can point.

Even 0-CFA analysis can be considerably more precise than Rapid Type Analysis. For example, in the program below, RTA would assume that any implementation of foo() could be invoked at any program location, but 0-CFA can distinguish the two call sites:

```
class A { A foo(A x) { return x; } }
class B extends A { A foo(A x) { return new D(); } }
class D extends A { A foo(A x) { return new A(); } }
class C extends A { A foo(A x) { return this; } }

// in main()
A x = new A();
while (...)
    x = x.foo(new B()); // may call A.foo, B.foo, or D.foo
A y = new C();
y.foo(x);               // only calls C.foo
```

Chapter 11

Axiomatic Semantics and Hoare-style Verification

> It has been found a serious problem to define these languages [ALGOL, FORTRAN, COBOL] with sufficient rigor to ensure compatibility among all implementations...One way to achieve this would be to insist that all implementations of the language shall satisfy the axioms and rules of inference which underlie proofs of properties of programs expressed in the language. In effect, this is equivalent to accepting the axioms and rules of inference as the ultimately definitive specification of the meaning of the language.
>
> *C.A.R Hoare, An Axiomatic Basis for Computer Programming,1969*

So far in this course we have largely represented and reasoned about programs (and analysis of those programs) in terms of operational semantics, which gives meaning to programs based on what happens when we execute them. Now, we turn our attention to a different kind of representation, which in turn enables a different kind of static reasoning about program correctness.

11.1 Axiomatic Semantics

Axiomatic semantics (or Hoare-style logic) defines the meaning of a statement in terms of its effects on assertions of truth that can be made about the associated program. This provides a formal system for reasoning about correctness. An axiomatic semantics fundamentally consists of: (1) a language for stating assertions about programs (where an assertion is something like "if this function terminates, $x > 0$ upon termination"), coupled with (2) rules for establishing the truth of assertions. Various logics have been used to encode such assertions; for simplicity, we will begin by focusing on first-order logic.

In this system, a *Hoare Triple* encodes such assertions:

$$\{P\}\ S\ \{Q\}$$

P is the precondition, Q is the postcondition, and S is a piece of code of interest. Relating this back to our earlier understanding of program semantics, this can be read as "if P holds in some state E and if $\langle E, S\rangle \Downarrow E'$, then Q holds in E'." We distinguish between partial ($\{P\}\ S\ \{Q\}$) and total ($[P]\ S\ [Q]$) correctness by saying that total correctness means that, given precondition P, S will terminate, and Q will hold; partial correctness does not make termination guarantees. We primarily focus on partial correctness.

11.1.1 Assertion judgements using operational semantics

Consider a simple assertion language adding first-order predicate logic to WHILE expressions:

$$\begin{array}{rcllll} P & ::= & \text{true} & \mid \text{false} & \mid e_1 = e_2 & \mid e_1 \geqslant e_2 \quad \mid P_1 \wedge P_2 \\ & \mid & P_1 \vee P_2 & \mid P_1 \Rightarrow P_2 & \mid \forall x.P & \mid \exists x.P \end{array}$$

Note that we are somewhat sloppy in mixing logical variables and program variables; all WHILE variables implicitly range over integers, and all WHILE boolean expressions are also assertions.

We now define an assertion judgement $E \models P$, read "P is true in E" or alternatively "E entails P." The $\models$ judgment is defined inductively on the structure of assertions, and relies on the operational semantics of WHILE arithmetic expressions. For example:

$$\begin{array}{ll} E \models \text{true} & \textit{always} \\ E \models a_1 = a_2 & \textit{iff } \langle E, a_1 \rangle \Downarrow n \text{ and } \langle E, a_2 \rangle \Downarrow n \\ E \models a_1 \geqslant a_2 & \textit{iff } \langle E, a_1 \rangle \Downarrow n_1, \langle E, a_2 \rangle \Downarrow n_2, \text{ and } n_1 \geqslant n_2 \\ E \models P_1 \wedge P_2 & \textit{iff } E \models P_1 \text{ and } E \models P_2 \\ \ldots & \\ E \models \forall x.P & \textit{iff } \forall n \in \mathbb{Z}.E[x \mapsto n] \models P \\ E \models \exists x.P & \textit{iff } \exists n \in \mathbb{Z}.E[x \mapsto n] \models P \end{array}$$

Now we can define formally the meaning of a partial correctness assertion $\models \{P\}\ S\ \{Q\}$:

$$\forall E. \forall E'. (E \models P \wedge \langle E, S \rangle \Downarrow E') \Rightarrow E' \models Q$$

Question: *What about total correctness?*

This gives us a formal, but unsatisfactory, mechanism to decide $\models \{P\}\ S\ \{Q\}$. By defining the judgement in terms of the operational semantics, we practically have to run the program to verify an assertion! It's also awkward/impossible to effectively verify the truth of a $\forall x.P$ assertion (check every integer?!). This motivates a new symbolic technique for deriving valid assertions from others that are known to be valid.

11.1.2 Derivation rules for Hoare triples

We write $\vdash P$ (read "we can prove P") when P can be derived from basic axioms. The derivation rules for $\vdash P$ are the usual ones from first-order logic with arithmetic, like (but obviously not limited to):

$$\frac{\vdash P \quad \vdash Q}{\vdash P \wedge Q}\ \textit{and}$$

We can now write $\vdash \{P\}\ S\ \{Q\}$ when we can derive a triple using derivation rules. There is one derivation rule for each statement type in the language (sound familiar?):

$$\frac{}{\vdash \{P\}\ \text{skip}\ \{P\}}\ \textit{skip} \qquad \frac{}{\vdash \{[a/x]P\}\ x{:=}a\ \{P\}}\ \textit{assign}$$

$$\frac{\vdash \{P\}\ S_1\ \{P'\} \qquad \vdash \{P'\}\ S_2\ \{Q\}}{\vdash \{P\}\ S_1;\ S_2\ \{Q\}}\ \textit{seq} \qquad \frac{\vdash \{P \wedge b\} S_1 \{Q\} \qquad \vdash \{P \wedge \neg b\}\ S_2\ \{Q\}}{\vdash \{P\}\ \text{if } b \text{ then } S_1 \text{ else } S_2\ \{Q\}}\ \textit{if}$$

Question: *What can we do for while?*

There is also the *rule of consequence*:

$$\frac{\vdash P' \Rightarrow P \qquad \vdash \{P\}\ S\ \{Q\} \qquad \vdash Q \Rightarrow Q'}{\vdash \{P'\}\ S\ \{Q'\}}\ consq$$

This rule is important because it lets us make progress even when the pre/post conditions in our program don't exactly match what we need (even if they're logically equivalent) or are stronger or weaker logically than ideal.

We can use this system to prove that triples hold. Consider $\{\text{true}\}\ x := e\ \{x = e\}$, using (in this case) the assignment rule plus the rule of consequence:

$$\frac{\vdash \text{true} \Rightarrow a = a \qquad \overline{\{a = a\}\ x := a\ \{x = a\}}}{\vdash \{\text{true}\} x := a \{x = a\}}$$

A system of axiomatic semantics is *sound* if everything we can prove is also true, that is: `if` $\vdash \{P\}S\{Q\}$ `then` $\models \{P\}S\{Q\}$. This can be proven via simultaneous induction on the structure of the operational semantics derivation and the axiomatic semantics proof; will not conduct this proof in these notes. Intuitively, it expresses that the axiomatic proof we can derive using these rules is equivalent to the operational semantics derivation

A system of axiomatic semantics is *complete* if we can prove all true things: `if` $\models \{P\}S\{Q\}$ `then` $\vdash \{P\}S\{Q\}$ The system we have outlined is relatively complete (that is, as complete as the underlying logic). We now move to showing how to (soundly/-completely) prove properties of programs using this style of semantics.

11.2 Proofs of a Program

Hoare-style verification is based on the idea of a specification as a contract between the implementation of a function and its clients. The specification consists of the precondition and a postcondition. The precondition is a predicate describing the condition the code/function relies on for correct operation; the client must fulfill this condition. The postcondition is a predicate describing the condition the function establishes after correctly running; the client can rely on this condition being true after the call to the function.

Note that if a client calls a function without fulfilling its precondition, the function can behave in any way at all and still be correct. Therefore, if a function must be robust to errors, the precondition should include the possibility of erroneous input, and the postcondition should describe what should happen in case of that input (e.g. an exception being thrown).

The goal in Hoare-style verification is thus to (statically!) prove that, given a pre-condition, a particular post-condition will hold after a piece of code executes. We do this by generating a logical formula known as a *verification condition*, constructed such that, if true, we know that the program behaves as specified. The general strategy for doing this, introduced by Dijkstra, relies on the idea of a *weakest precondition* of a statement with respect to the desired post-condition. We then show that the given precondition implies it. However, loops, as ever, complicate this strategy.

11.2.1 Strongest postconditions and weakest pre-conditions

We can write any number of perfectly valid Hoare triples. Consider the Hoare triple $\{x = 5\}\ x := x * 2\ \{x > 0\}$. This triple is clearly correct, because if $x = 5$ and we multiply x by 2, we get $x = 10$ which clearly implies that $x > 0$. However, although correct, this Hoare triple is not a precise as we might like. Specifically, we could write a stronger postcondition, i.e. one that implies $x > 0$. For example, $x > 5 \wedge x < 20$ is stronger because it is more informative;

it pins down the value of x more precisely than $x > 0$. The strongest postcondition possible is $x = 10$; this is the most useful postcondition. Formally, if $\{P\}\ S\ \{Q\}$ and for all Q' such that $\{P\}\ S\ \{Q'\}$, $Q \Rightarrow Q'$, then Q is the strongest postcondition of S with respect to P.

We can compute the strongest postcondition for a given statement and precondition using the function $sp(S, P)$. Consider the case of a statement of the form $x := e$. If the condition P held before the statement, we now know that P still holds and that $x = e$—where P and e are now in terms of the old, pre-assigned value of x. For example, if P is $x + y = 5$, and S is $x := x + z$, then we should know that $x' + y = 5$ and $x = x' + z$, where x' is the old value of x. The program semantics doesn't keep track of the old value of x, but we can express it by introducing a fresh, existentially quantified variable x'. This gives us the following strongest postcondition for assignment:[1]

$$sp(x := a, P) \quad = \exists x'.[x'/x]P \wedge x = [x'/x]a$$

While this scheme is workable, it is awkward to existentially quantify over a fresh variable at every statement; the formulas produced become unnecessarily complicated, and if we want to use automated theorem provers, the additional quantification tends to cause problems. Dijkstra proposed reasoning instead in terms of *weakest preconditions*, which turns out to work better. If $\{P\}\ S\ \{Q\}$ and for all P' such that $\{P'\}\ S\ \{Q\}$, $P' \Rightarrow P$, then P is the weakest precondition $wp(S, Q)$ of S with respect to Q.

We can define a function yielding the weakest precondition inductively, following the Hoare rules. For for assignments, sequences, and if statements, this yields:

$$\begin{aligned}
&wp(x := a, P) && = [a/x]P \\
&wp(S;\ T, Q) && = wp(S,\ wp(T, Q)) \\
&wp(\text{if } b \text{ then } S \text{ else } T, Q) && = b \Rightarrow wp(S, Q) \wedge \neg b \Rightarrow wp(T, Q)
\end{aligned}$$

11.2.2 Loops

As usual, things get tricky when we get to loops. Consider:

$$\{P\}\ \text{while}(i < x)\ \text{do}\ f = f * i; i := i + 1\ \ \text{done}\{f = x!\}$$

What is the weakest precondition here? Fundamentally, we need to prove by induction that the property we care about will generalize across an arbitrary number of loop iterations. Thus, P is the base case, and we need some inductive hypothesis that is preserved when executing loop body an arbitrary number of times. We commonly refer to this hypothesis as a *loop invariant*, because it represents a condition that is always true (i.e. invariant) before and after each execution of the loop.

Computing weakest preconditions on loops is very difficult on real languages. Instead, we assume the provision of that loop invariant. A loop invariant must fulfill the following conditions:

- $P \Rightarrow I$: The invariant is initially true. This condition is necessary as a base case, to establish the induction hypothesis.
- $\{Inv \wedge b\}\ S\ \{Inv\}$: Each execution of the loop preserves the invariant. This is the inductive case of the proof.

[1] Recall that the operation $[x'/x]e$ denotes the capture-avoiding substitution of x' for x in e; we rename bound variables as we do the substitution so as to avoid conflicts.

- $(Inv \wedge \neg b) \Rightarrow Q$: The invariant and the loop exit condition imply the postcondition. This condition is simply demonstrating that the induction hypothesis/loop invariant we have chosen is sufficiently strong to prove our postcondition Q.

The procedure outlined above only verifies partial correctness, because it does not reason about how many times the loop may execute. Verifying full correctness involves placing an upper bound on the number of remaining times the loop body will execute, typically called a *variant function*, written v, because it is variant: we must prove that it decreases each time we go through the loop. We mention this for the interested reader; we will not spend much time on it.

11.2.3 Proving programs

Assume a version of WHILE that annotates loops with invariants: while$_{inv}$ b do S. Given such a program, and associated pre- and post-conditions:

$$\{P\}\ S\ \{Q\}$$

The proof strategy constructs a verification condition $VC(S_{annot}, Q)$ that we seek to prove true (usually with the help of a theorem prover). VC is guaranteed to be stronger than $wp(S, Q)$ but still weaker than P: $P \Rightarrow VC(S, Q) \Rightarrow wp(S, Q)$ We compute VC using a verification condition generation procedure $VCGen$, which mostly follows the definition of the wp function discussed above:

$$\begin{array}{lcl} VCGen(\text{skip}, Q) & = & Q \\ VCGen(S_1; S_2, Q) & = & VCGen(S_1, VCGen(S_2, Q)) \\ VCGen(\text{if } b \text{ then } S_1 \text{ else } S_2, Q) & = & b \Rightarrow VCGen(S_1, Q) \wedge \neg b \Rightarrow VCGen(S_2, Q) \\ VCGen(x := a, Q) & = & [a/x]Q \end{array}$$

The one major point of difference is in the handling of loops:

$$VCGen(\text{while}_{inv}\ b \text{ do } S, Q) = Inv \wedge (\forall x_1 ... x_n . Inv \Rightarrow (b \Rightarrow VCGen(S, Inv) \wedge \neg b \Rightarrow Q))$$

To see this in action, consider the following WHILE program:

$$\begin{array}{l} r := 1; \\ i := 0; \\ \text{while } i < m \text{ do} \\ \qquad r := r * n; \\ \qquad i := i + 1 \end{array}$$

We wish to prove that this function computes the nth power of m and leaves the result in r. We can state this with the postcondition $r = n^m$.

Next, we need to determine a precondition for the program. We cannot simply compute it with wp because we do not yet know the loop invariant is—and in fact, different loop invariants could lead to different preconditions. However, a bit of reasoning will help. We must have $m \geqslant 0$ because we have no provision for dividing by n, and we avoid the problematic computation of 0^0 by assuming $n > 0$. Thus our precondition will be $m \geqslant 0 \wedge n > 0$.

A good heuristic for choosing a loop invariant is often to modify the postcondition of the loop to make it depend on the loop index instead of some other variable. Since the loop index runs from i to m, we can guess that we should replace m with i in the postcondition $r = n^m$. This gives us a first guess that the loop invariant should include $r = n^i$.

This loop invariant is not strong enough, however, because the loop invariant conjoined with the loop exit condition should imply the postcondition. The loop exit condition is $i \geqslant m$, but we need to know that $i = m$. We can get this if we add $i \leqslant m$ to the loop invariant. In addition, for proving the loop body correct, we will also need to add $0 \leqslant i$ and $n > 0$ to the loop invariant. Thus our full loop invariant will be $r = n^i \wedge 0 \leqslant i \leqslant m \wedge n > 0$.

Our next task is to use weakest preconditions to generate proof obligations that will verify the correctness of the specification. We will first ensure that the invariant is initially true when the loop is reached, by propagating that invariant past the first two statements in the program:

$$\begin{array}{l}\{m \geqslant 0 \wedge n > 0\} \\ r := 1; \\ i := 0; \\ \{r = n^i \wedge 0 \leqslant i \leqslant m \wedge n > 0\}\end{array}$$

We propagate the loop invariant past $i := 0$ to get $r = n^0 \wedge 0 \leqslant 0 \leqslant m \wedge n > 0$. We propagate this past $r := 1$ to get $1 = n^0 \wedge 0 \leqslant 0 \leqslant m \wedge n > 0$. Thus our proof obligation is to show that:

$$m \geqslant 0 \wedge n > 0 \Rightarrow 1 = n^0 \wedge 0 \leqslant 0 \leqslant m \wedge n > 0$$

We prove this with the following logic:

$m \geqslant 0 \wedge n > 0$	by assumption
$1 = n^0$	because $n^0 = 1$ for all $n > 0$ and we know $n > 0$
$0 \leqslant 0$	by definition of $\leqslant$
$0 \leqslant m$	because $m \geqslant 0$ by assumption
$n > 0$	by the assumption above
$1 = n^0 \wedge 0 \leqslant 0 \leqslant m \wedge n > 0$	by conjunction of the above

To show the loop invariant is preserved, we have:

$$\begin{array}{l}\{r = n^i \wedge 0 \leqslant i \leqslant m \wedge n > 0 \wedge i < m\} \\ r := r * n; \\ i := i + 1; \\ \{r = n^i \wedge 0 \leqslant i \leqslant m \wedge n > 0\}\end{array}$$

We propagate the invariant past $i := i + 1$ to get $r = n^{i+1} \wedge 0 \leqslant i + 1 \leqslant m \wedge n > 0$. We propagate this past $r := r * n$ to get: $r * n = n^{i+1} \wedge 0 \leqslant i + 1 \leqslant m \wedge n > 0$. Our proof obligation is therefore:

$$\begin{array}{l}r = n^i \wedge 0 \leqslant i \leqslant m \wedge n > 0 \wedge i < m \\ \Rightarrow r * n = n^{i+1} \wedge 0 \leqslant i + 1 \leqslant m \wedge n > 0\end{array}$$

We can prove this as follows:

$r = n^i \wedge 0 \leqslant i \leqslant m \wedge n > 0 \wedge i < m$	by assumption
$r * n = n^i * n$	multiplying by n
$r * n = n^{i+1}$	definition of exponentiation
$0 \leqslant i + 1$	because $0 \leqslant i$
$i + 1 < m + 1$	by adding 1 to inequality
$i + 1 \leqslant m$	by definition of $\leqslant$
$n > 0$	by assumption
$r * n = n^{i+1} \wedge 0 \leqslant i + 1 \leqslant m \wedge n > 0$	by conjunction of the above

Last, we need to prove that the postcondition holds when we exit the loop. We have already hinted at why this will be so when we chose the loop invariant. However, we can state the proof obligation formally:

$$r = n^i \wedge 0 \leqslant i \leqslant m \wedge n > 0 \wedge i \geqslant m$$
$$\Rightarrow r = n^m$$

We can prove it as follows:

$r = n^i \wedge 0 \leqslant i \leqslant m \wedge n > 0 \wedge i \geqslant m$	by assumption
$i = m$	because $i \leqslant m$ and $i \geqslant m$
$r = n^m$	substituting m for i in assumption

Chapter 12

Satisfiability Modulo Theories

12.1 Motivation and Overview

Program analysis often makes use of techniques that generate and solve logical formulas. For example, in Hoare-style verification, we used weakest preconditions and verification conditions to generate formulas of the form $P \Rightarrow Q$. Usually P and Q have free variables x, e.g. P could be $x > 3$ and Q could be $x > 1$. We want to prove, ideally automatically, that $P \Rightarrow Q$ no matter what x we choose, i.e. no matter what the model (an assignment from variables to values) is. This is equivalent to saying $P \Rightarrow Q$ is *valid*. We will see in subsequent chapters how symbolic and concolic execution similarly generate sets of *patch conditions* with free variables, such as those that correspond to program inputs. We would like to determine if such path conditions are feasible; we moreover would like to identify values for those free variables, because that can help generate test inputs that cover particular program paths. SMT solving addresses this type of problem. Although the general goal won't be feasible for all formulas, it is feasible for a useful subset of formulas.

Solving this problem begins by reducing general formula validity to another problem, that of *satisfiability*. A formula F with free variable x is valid iff for all x, F is true. That's the same thing as saying there is no x for which F is false. But that's furthermore the same as saying there is no x for which $\neg F$ is true. This last formulation is asking whether $\neg F$ is *satisfiable*. It turns out to be easier to seach for a single satisfying model (or prove there is none), then to show that a formula is valid for all models.

Strictly speaking, satisfiability is for boolean formulas, or those that include boolean variables as well as boolean operators such as $\wedge$, $\vee$, and $\neg$. They may include quantifiers such as $\forall$ and $\exists$, as well. But if we want to have variables over the integers or reals, and operations over numbers (e.g. $+, >$, the types of relations we've included even in our very simple WHILE language), we need a solver for a *theory*, such as the theory of Presburger arithmetic (which could prove that $2 * x = x + x$), or the theory of arrays (which could prove that assigning $x[y]$ to 3 and then looking up $x[y]$ yields 3). SMT solvers include a basic satisfiability checker, and allow that checker to communicate with specialized solvers for those theories. This is the meaning of the "Modulo Theories" in "Satisfiability Modulo Theories."

12.2 DPLL for Boolean Satisfiability

In building to satisfiability modulo theories, we begin by discussing the problem of satisfiability.

12.2.1 Boolean satisfiability (SAT)

Satisfiability decides whether a conjunction of literals in a theory is satisfiable. The "easiest" theory is propositional logic. The problem of establishing satisfiability for boolean formulas in propositional logic is referred to as SAT, and a decision procedure for it as a "SAT solver."

We begin by transforming a formula F into *conjuctive normal form (CNF)*—i.e. a conjunction of disjunctions of positive or negative literals. For example $(a \vee \neg b) \wedge (\neg a \vee c) \wedge (b \vee c)$ is a CNF formula. If the formula is not already in CNF, we can put it into CNF by using De Morgan's laws, the double negative law, and the distributive laws:

$$\begin{array}{rcl} \neg(P \vee Q) & \iff & \neg P \wedge \neg Q \\ \neg(P \wedge Q) & \iff & \neg P \vee \neg Q \\ \neg\neg P & \iff & P \\ (P \wedge (Q \vee R)) & \iff & ((P \wedge Q) \vee (P \wedge R)) \\ (P \vee (Q \wedge R)) & \iff & ((P \vee Q) \wedge (P \vee R)) \end{array}$$

The goal of the decision procedure is, given a formula, to say it is satisfiable; this can be established by giving an example *satisfying assignment*. A satisfying assignment maps variables to boolean values. So, $X \vee Y$ is satisfiable, and one satisfying assignment for it is $X \mapsto \texttt{true}, Y \mapsto \texttt{false}$ (there are other satisfying assignments as well). $X \wedge \neg X$, by contrast, is not satisfiable.

The Cook-Levin theorem established that boolean satisfiability is NP-complete. In the worst case, one can decide SAT for a given formula by simply trying all possible assignments. For example, given:

$$\exists E. E \models (x \vee y \vee \neg z) \wedge (\neg x \vee \neg y) \wedge (z)$$

We can brute-force by conducting a backtracking search that tries all possible assignments of `true` and `false` to x, y, z. There are 2^n possible combinations in the worst case, where n is the number of variables.

12.2.2 The DPLL Algorithm

The DPLL algorithm, named for its developers Davis, Putnam, Logemann, and Loveland, is an efficient approach to deciding SAT. The DPLL algorithm improves on the backtracking search with two innovations: *unit propagation*, and *pure literal elimination*.

Let's illustrate by example. Consider the following formula:

$$(b \vee c) \wedge (\mathbf{a}) \wedge (\neg a \vee c \vee d) \wedge (\neg c \vee d) \wedge (\neg c \vee \neg d \vee \neg a) \wedge (b \vee d)$$

There is one clause with just a in it. This clause, like all other clauses, has to be true for the whole formula to be true, so we must make a true for the formula to be satisfiable. We can do this whenever we have a clause with just one literal in it, i.e. a unit clause. (Of course, if a clause has just $\neg b$, that tells us b must be false in any satisfying assignment). In this example, we use the *unit propagation* rule to replace all occurrences of a with `true`. After simplifying, this gives us:

$$(\mathbf{b} \vee c) \wedge (c \vee d) \wedge (\neg c \vee d) \wedge (\neg c \vee \neg d) \wedge (\mathbf{b} \vee d)$$

Now here we can see that b always occurs positively (i.e. without a $\neg$ in front of it). If we set b to be `true`, that eliminates all occurrences of b from our formula, thereby making it simpler—but it doesn't change the satisfiability of the underlying formula. An analogous approach applies when a variable always occurs negatively. A literal that occurs only positively, or only negatively, in a formula is called *pure*. Therefore, this simplification is called the *pure literal elimination* rule, and applying it to the example above gives us:

$$(c \vee d) \wedge (\neg c \vee d) \wedge (\neg c \vee \neg d)$$

Now for this formula, neither of the above rules applies. We just have to pick a literal and guess its value. Let's pick c and set it to `true`. Simplifying, we get:

$$(d) \wedge (\neg d)$$

After applying the unit propagation rule (setting d to true) we get:

$$(\texttt{true}) \wedge (\texttt{false})$$

This didn't work out! But remember, we guessed about the value of c. Let's backtrack to the formula where we made that choice:

$$(c \vee d) \wedge (\neg c \vee d) \wedge (\neg c \vee \neg d)$$

and now we'll try things the other way, i.e. with $c = \texttt{false}$. Then we get the formula

$$(d)$$

because the last two clauses simplified to true once we know c is false. Now unit propagation sets $d = \texttt{true}$ and then we have shown the formula is satisfiable. A real DPLL algorithm would keep track of all the choices in the satisfying assignment, and would report back that a is true, b is true, c is false, and d is true in the satisfying assignment.

This procedure—applying unit propagation and pure literal elimination eagerly, then guessing a literal and backtracking if the guess goes wrong—is the essence of DPLL. Here's an algorithmic statement of DPLL, adapted slightly from a version on Wikipedia:

function DPLL(ϕ)
  if $\phi = \texttt{true}$ **then**
    return `true`
  end if
  if ϕ contains a `false` clause **then**
    return `false`
  end if
  for all unit clauses l in ϕ **do**
    $\phi \leftarrow$ UNIT-PROPAGATE(l, ϕ)
  end for
  for all literals l occurring pure in ϕ **do**
    $\phi \leftarrow$ PURE-LITERAL-ASSIGN(l, ϕ)
  end for
  $l \leftarrow$ CHOOSE-LITERAL(ϕ)
  return DPLL($\phi \wedge l$) $\vee$ DPLL($\phi \wedge \neg l$)
end function

Mostly the algorithm above is straightforward, but there are a couple of notes. First, why does the algorithm do unit propagation before pure literal assignment? It's good to do unit propagation first because it can create additional opportunities to apply further unit propagation as well as pure literal assignment. On the other hand, pure literal assignment will never create unit literals that didn't exist before: pure assignment can eliminate entire clauses, but never makes an existing clause shorter.

Secondly, the last line implements backtracking. We assume a short-cutting $\vee$ operation at the level of the algorithm. So if the first recursive call to DPLL returns true, so does the current call–but if it returns fall, we invoke DPLL with the chosen literal negated, which effectively backtracks.

Exercise 1. Apply DPLL to the following formula, describing each step (unit propagation, pure literal elimination, choosing a literal, or backtracking) and showing now it affects the formula until you prove that the formula is satisfiable or not:

$$(a \vee b) \wedge (a \vee c) \wedge (\neg a \vee c) \wedge (a \vee \neg c) \wedge (\neg a \vee \neg c) \wedge (\neg d)$$

There is a lot more to learn about DPLL, including hueristics for how to choose the literal l to be guessed and smarter approaches to backtracking (e.g. non-chronological backtracking), but in this class, let's move on to consider SMT.

12.3 Solving SMT Problems

The approach above targets formulas in the theory of propositional logic. However, there are many other possibly useful theories, and useful formulas may mix them. For example, consider a conjunction of the following formulas:[1]

$$\begin{array}{ll} f(f(x) - f(y)) = a & \wedge \\ f(0) = a + 2 & \wedge \\ x = y & \end{array}$$

This problem mixes linear arithmetic with the theory of uninterpreted functions (here, f is some unknown function). We may have a satisfiability procedure for each theory involved in the formula, but how can we deal with their combination? Note that we *can't* in general just separate out the terms from each theory in a formula to see if they are separately satisfiable, because multiple satisfying assignments might not be compatible. Instead, we handle each domain separately (as a theory), and then combine them all together using DPLL and SAT as the "glue".

12.3.1 Definitions

A *satisfiability modulo theories (SMT)* solver operates on propositions involving both logical terms and terms from theories (defined below). Effectively, such solvers replace all the theory clauses in a mixed-theory formula with special propositional variables, and then use a pure SAT solver to solve the result. If the solution involves any of the theory clauses, the solver asks the theory if they can all be true. If not, new constraints are added to the formula, and the process repeats.

In general, a *theory* is a set of sentences (syntax) with a deductive system that can determine satisfiability (semantics). Usually, the set of sentences is formally defined by a grammar of terms over atoms. The satisfying assignment (or model, or interpretation) maps literals (terms or negated terms) to booleans. Useful theories include linear and non-linear arithmetic, bitvectors, arrays, quantifiers, or strings, as well as uninterpreted functions (like f in our example above).

An important feature of the kinds of theories we are discussing is that they all understand *equality*. We do not delve deeply into how/why in these notes, but the fact is important to how SMT can solve formulas that mix theories, as we will see below.

12.3.2 Basic SMT idea, illustrated

We will work through our example above to demonstrate the ideas behind SMT. The first step is to separate the multiple theories. We can do this by replacing expressions with fresh variables, in a procedure named Nelson-Oppen after its two inventors. For example, in the

[1]This example is due to Oliveras and Rodriguez-Carbonell

first formula, we'd like to factor out the subtraction, so we generate a fresh variable and divide the formula into two:

$$\begin{array}{ll} f(e1) = a & \textit{// in the theory of uninterpreted functions now} \\ e1 = f(x) - f(y) & \textit{// still a mixed formula} \end{array}$$

Now we want to separate out $f(x)$ and $f(y)$ as variables $e2$ and $e3$, so we get:

$$\begin{array}{ll} e1 = e2 - e3 & \textit{// in the theory of arithmetic now} \\ e2 = f(x) & \textit{// in the theory of uninterpreted functions} \\ e3 = f(y) & \textit{// in the theory of uninterpreted functions} \end{array}$$

We can do the same for $f(0) = a + 2$, yielding:

$$\begin{array}{l} f(e4) = e5 \\ e4 = 0 \\ e5 = a + 2 \end{array}$$

We now have formulas in two theories. First, formulas in the theory of uninterpreted functions:

$$\begin{array}{l} f(e1) = a \\ e2 = f(x) \\ e3 = f(y) \\ f(e4) = e5 \\ x = y \end{array}$$

And second, formulas in the theory of arithmetic:

$$\begin{array}{l} e1 = e2 - e3 \\ e4 = 0 \\ e5 = a + 2 \\ x = y \end{array}$$

Notice that $x = y$ is in both sets of formulas (remember, all theories understand equality). First, however, let's run a solver. The solver for uninterpreted functions has a congruence closure rule that states, for all f, x, and y, if $x = y$ then $f(x) = f(y)$. Applying this rule (since $x = y$ is something we know), we discover that $f(x) = f(y)$. Since $f(x) = e2$ and $f(y) = e3$, by transitivity we know that $e2 = e3$.

But $e2$ and $e3$ are symbols that the arithmetic solver knows about, so we add $e2 = e3$ to the set of formulas we know about arithmetic. Now the arithmetic solver can discover that $e2 - e3 = 0$, and thus $e1 = e4$. We communicate this discovered equality to the uninterpreted functions theory, and then we learn that $a = e5$ (again, using congruence closure and transitivity).

This fact goes back to the arithmetic solver, which evaluates the following constraints:

$$\begin{array}{l} e1 = e2 - e3 \\ e4 = 0 \\ e5 = a + 2 \\ x = y \\ e2 = e3 \\ a = e5 \end{array}$$

Now there is a contradiction: $a = e5$ but $e5 = a + 2$. That means the original formula is unsatisfiable.

In this case, one theory was able to infer equality relationships that another theory could directly use. But sometimes a theory doesn't figure out an equality relationship, but only

certain correlations, e.g., e1 is either equal to e2 or e3. In the more general case, we can simply generate a formula that represents all possible equalities between shared symbols, which would look something like:

$$(e1 = e2 \vee e1 \neq e2) \wedge (e2 = e3 \vee e2 \neq e3) \wedge (e1 = e3 \vee e1 \neq e3) \wedge ...$$

We can now look at all possible combinations of equalities. In fact, we can use DPLL to do this, and DPLL also explains how we can combine expressions in the various theories with boolean operators such as $\wedge$ and $\vee$. If we have a formula such as:[2]

$$x \geqslant 0 \wedge y = x + 1 \wedge (y > 2 \vee y < 1)$$

We can then convert each arithmetic (or uninterpreted function) formula into a fresh propositional symbol, to get:

$$p1 \wedge p2 \wedge (p3 \vee p4)$$

and then run a SAT solver using DPLL. DPLL will return a satisfying assignment, such as $p1, p2, \neg p3, p4$. We then check this against each of the theories. In this case, the theory of arithmetic finds a contradiction: $p1$, $p2$, and $p4$ can't all be true, because $p1$ and $p2$ together imply that $y \geqslant 1$. We add a clause saying that these can't all be true and give it back to the SAT solver:

$$p1 \wedge p2 \wedge (p3 \vee p4) \wedge (\neg p1 \vee \neg p2 \vee \neg p3)$$

Running DPLL again gives us $p1, p2, p3, \neg p4$. We check this against the theory of arithmetic, and it all works out.

12.3.3 DPLL(T)

This use of DPLL parameterized with respect to a set of theories T is called DPLL(T) or (DPLL-T), and it is an SMT algorithm. At a high level, DPLL(T) works as we illustrated above: it converts mixed constraints to boolean constraints and then runs DPLL, and then checks the resulting assignments with the underlying theories to determine if they are valid. The version of DPLL used in DPLL(T) has two key changes compared to the original, however.

First, DPLL(T) does not use the pure variable elimination optimization. This is because, in pure propositional logic, variables are necessarily independent. If some variable x only appears positively, you can set it to `true` and save time. With theories, variables may be dependent. For example, consider

$$(\mathbf{x > 10} \vee x < 3) \wedge (\mathbf{x > 10} \vee x < 9) \wedge (x < 7)$$

In the above, $x > 10$, but if we just set that term to be true as part of the model, the other terms all become false. We cannot simply skip over it.

Second, unit propagation interacts with the theories to add constraints to the formula when available. We saw this example above, when the theory of arithmetic found a contraditiction in the formula

$$p1 \wedge p2 \wedge (p3 \vee p4)$$

And the solving procedure added a clause to the formula before giving it back to the SAT solver.

[2]If we had multiple theories, I am assuming we've already added the equality constraints between them, as described above.

12.3.4 Bonus: Arithmetic solvers

We discussed above how the solver for the theory of uninterpreted functions work; how does the arithmetic solver work? In cases like the above example where we assert formulas of the form $y = x + 1$, we can eliminate y by substituting it with $x + 1$ everywhere. In the cases where we only constrain a variable using inequalities, there is a more general approach called Fourier-Motzkin Elimination. In this approach, we take all inequalities that involve a variable x and transform them into one of the following forms:

$$A \leqslant x$$
$$x \leqslant B$$

where A and B are linear formulas that don't include x. We can then eliminate x, replacing the above formulas with the equation $A \leqslant B$. If we have multiple formulas with x on the left and/or right, we just conjoin the cross product. There are various optimizations that are applied in practice, but the basic algorithm is general and provides a broad understanding of how arithmetic solvers work.

Chapter 13

Symbolic Execution

13.1 Overview

In previous lectures, we developed axiomatic semantics as a way to represent a program meaning in terms of what is true after code executes. We used weakest preconditions (versus strongest postconditions) to prove program properties (that is, that some property Q is true after S executes, assuming P is true before it executes). However, because weakest preconditions are typically infeasible to compute, we further extended our language to include loop invariants, and used them to compute *verification conditions* to verify program properties.

13.1.1 Forward Verification Condition Intuition

Revisiting the verification condition generation procedure, recall that it effectively works *backwards* (which should make sense, given our intuition regarding weakest preconditions). This is visible in the way the recursive calls to $VCGen$ worked. Consider computing the VC for a sequence of assignments with respect to some post condition Q:

$$\begin{array}{l}\{P\}\\ x := e_1\\ x := e_2\\ \{Q\}\end{array}$$

But what if instead, we went *forwards*, after all? We could get a proof obligation of the form:

$$\forall x_n : ([x_0/x]P \wedge x_1 = [x_0/x]e_1 \wedge x_2 = ([x_1/x]e_2)) \Rightarrow [x_2/x]Q$$

We use fresh mathematical variables x_i each time the value of the mutable program var `x` is updated. For example, to prove the following:

$$\begin{array}{l}\{x > 0\}\\ x := x * 2\\ x := x + 1\\ \{x > 1\}\end{array}$$

we need to show that:

$$\forall x_0, x_1, x_2 \in \mathbb{Z} : (x_0 > 0 \wedge x_1 = x_0 * 2 \wedge x_2 = x_1 + 1) \Rightarrow x_2 > 1$$

Computing a verification conditions forwards is conducted using a technique known as *symbolic execution*. Symbolic execution is a general technique that executes a program abstractly, so that one abstract execution covers multiple possible inputs of the program that share a particular execution path through the code. We introduce it as a mechanism to

generate verification conditions; one benefit of symbolic execution is that it makes axiomatic semantics practical. However, it can also be viewed as a way to generalize testing, automate testing, and find bugs in programs (as we will see).

Things get a bit more complicated when dealing with conditional branching. Consider:

$$
\begin{array}{l}
\{true\} \\
\textbf{if}\,(x < 0): \\
\qquad y := -x \\
\textbf{else}: \\
\qquad y := x \\
\{y \geqslant 0\}
\end{array}
$$

One option is to traverse each program path independently and generate distinct verification conditions. If all of them can be proven to be true, then the program is correct:

$$\forall x_0, y_0 \in \mathbb{Z} : (x_0 < 0 \wedge y_0 = -x_0) \Rightarrow y_0 \geqslant 0$$

$$\forall x_0, y_0 \in \mathbb{Z} : (x_0 \geqslant 0 \wedge y_0 = x_0) \Rightarrow y_0 \geqslant 0$$

Another option is to combine these conditions into one formula by merging information at join points; that is, when the two branches of the if-else come together. Verification reduces to showing that the following formula is true:

$$\forall x_0, y_0 \in \mathbb{Z} : ((x_0 < 0 \Rightarrow y_0 = -x_0) \vee (x_0 \geqslant 0 \Rightarrow y_0 = x_0)) \Rightarrow y_0 \geqslant 0$$

The former strategy is often referred to as *dynamic symbolic execution* (DSE) while the latter is *static symbolic execution* (SSE) [2, 31]. The advantage of SSE is that formulas can often be made compact by removing redundancies (e.g. for variables that don't get modified along either conditional branch) and making other algebraic simplifications across paths. However, DSE retains provenance; if any of the verification conditions don't hold true, then it is easy to determine the execution path along which an error occurs (and find a corresponding input). Further, DSE lends itself well to performing under-approximate analysis when dealing external library calls and loops with symbolic bounds.

13.1.2 Formalizing Forward VCGen

We can define forward verification condition generation in terms of symbolically evaluating the program, using the same kinds of big-step rules we used for operational semantics. This makes it easier to understand how verification conditions propagate along multiple program paths and uses symbolic environments to track the formulas for mutable variables (which mechanizes the generation of "fresh" mathematical variables).

Symbolic expressions. We start by defining symbolic analogs for arithmetic expressions and boolean predicates. We will call symbolic predicates *guards* and use the metavariable g, as these will turn into guards for paths the symbolic evaluator explores. These analogs are the same as the ordinary versions, except that in place of variables we use symbolic constants:

$$
\begin{array}{rcl@{\qquad}rcl}
g & ::= & \text{true} & a_s & ::= & \alpha \\
 & | & \text{false} & & | & n \\
 & | & \text{not } g & & | & a_{s1}\ op_a\ a_{s2} \\
 & | & g_1\ op_b\ g_2 & & & \\
 & | & a_{s1}\ op_r\ a_{s2} & & &
\end{array}
$$

Next, we generalize the notion of the environment Σ, so that variables refer not just to integers but to symbolic expressions:

$$\Sigma \in \mathit{Var} \to a_s$$

Big-step rules for the symbolic evaluation of expressions results in symbolic expressions. Since we don't have actual values in many cases, the expressions won't evaluate, but variables will be replaced with symbolic constants. That is, we obtain a symbolic expression from a concrete expression e by replacing all variables x in e with their values in the current symbolic state Σ. When these values are concrete, we use the concrete values; if not, we replace variables with symbolic constants:

$$\frac{}{\langle \Sigma, n \rangle \Downarrow n} \textit{ big-int}$$

$$\frac{}{\langle \Sigma, x \rangle \Downarrow \Sigma(x)} \textit{ big-var}$$

$$\frac{\langle \Sigma, a_1 \rangle \Downarrow a_{s1} \quad \langle \Sigma, a_2 \rangle \Downarrow a_{s2}}{\langle \Sigma, a_1 + a_2 \rangle \Downarrow a_{s1} + a_{s2}} \textit{ big-add}$$

Dynamic symbolic execution (DSE) of statements. We can likewise define rules for statement evaluation. These rules update not only the environment Σ, but also a path guard g associated with that state:

$$\frac{}{\langle g, \Sigma, \texttt{skip} \rangle \Downarrow \langle g, \Sigma \rangle} \textit{ big-skip}$$

$$\frac{\langle g, \Sigma, s_1 \rangle \Downarrow \langle g', \Sigma' \rangle \quad \langle g', \Sigma', s_2 \rangle \Downarrow \langle g'', \Sigma'' \rangle}{\langle g, \Sigma, s_1; s_2 \rangle \Downarrow \langle g'', \Sigma'' \rangle} \textit{ big-seq}$$

$$\frac{\langle \Sigma, a \rangle \Downarrow a_s}{\langle g, \Sigma, x := a \rangle \Downarrow \langle g, \Sigma[x \mapsto a_s] \rangle} \textit{ big-assign}$$

$$\frac{\langle \Sigma, b \rangle \Downarrow g' \quad g \wedge g' \texttt{SAT} \quad \langle g \wedge g', \Sigma, s_1 \rangle \Downarrow \langle g'', \Sigma' \rangle}{\langle g, \Sigma, \texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2, \rangle \Downarrow \langle g'', \Sigma' \rangle} \textit{ big-iftrue}$$

$$\frac{\langle \Sigma, b \rangle \Downarrow g' \quad g \wedge \neg g' \texttt{SAT} \quad \langle g \wedge \neg g', \Sigma, s_2 \rangle \Downarrow \langle g'', \Sigma' \rangle}{\langle g, \Sigma, \texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2, \rangle \Downarrow \langle g'', \Sigma' \rangle} \textit{ big-iffalse}$$

The rules for `skip`, sequence, and assignment are compositional in the expected way. The rules for `if` are more interesting. Here, we evaluate the condition to a symbolic predicate g'. In the true case, we use a SMT solver to verify that the guard is satisfiable when conjoined with the existing path condition. If that's the case, we continue by evaluating the true branch symbolically. The false case is analogous.

Doing this produces a symbolic state and a path condition or guard that describes that state. In practice, however, multiple states or paths are possible given symbolic inputs, and so in the general case, dynamic symbolic execution in fact produces a *tree* of possible paths/symbolic states guarded by path conditions that lead to them.

Static symbolic execution (SSE) of statements As noted earlier, SSE merges symbolic formulas at join points. We won't write out all the rules for SSE here, but most of them are similar to that of DSE with the main exception of `big-seq` which handles the sequence of statements $s_1; s_2$. For static symbolic execution, we would consider *all possible* $\langle g', \Sigma' \rangle$ such that $\langle g, \Sigma, s_1 \rangle \Downarrow \langle g', \Sigma' \rangle$, and merge these symbolic environments together before propagating them to s_2. How does merging work? Consider the merge of $\langle g_1, \Sigma_1 \rangle$ with $\langle g_2, \Sigma_1 \rangle$ to result in Σ_{merged}. The merging introduces fresh symbolic variables α_x corresponding to each x such that $\Sigma_1(x) \neq \Sigma_2(x)$; then sets $\Sigma_{merged}(x) = \alpha_x$; and then adds to the path constraints $(g_1 \Rightarrow \alpha_x = \Sigma_1(x)) \vee (g_2 \Rightarrow \alpha_x = \Sigma_2(x))$ to the original constraint g. All other mappings that are common in Σ_1 and Σ_2 are propagated as-is in Σ_{merged}.

Loops. Speaking of multiple paths: what about loops? The simplest way to handle `while` is analogous to `if`, above, but this will lead to a potentially infinite number of paths if the loop conditions contain symbolic formulas that are always feasible. Practical DSE tools (such as what is used in industry, discussed below) typically choose to symbolically "execute" loops up to k times (where k is usually no more than 1 or 2). This approach is particularly applicable if the goal is to find bugs or symbolically enumerate as many paths through the program as possible, especially on real-world code without loop invariants.

$$\frac{k > 0 \quad \langle \Sigma, b \rangle \Downarrow g' \quad g \wedge g' \texttt{SAT} \quad \langle g \wedge g', \Sigma, s; \texttt{while}_{\texttt{k}-1}\ b\ \texttt{do}\ s \rangle \Downarrow \langle g'', \Sigma' \rangle}{\langle g, \Sigma, \texttt{while}_{\texttt{k}}\ b\ \texttt{do}\ s, \rangle \Downarrow \langle g'', \Sigma' \rangle}\ \textit{big-whiletrue}$$

$$\frac{\langle \Sigma, b \rangle \Downarrow g' \quad g \wedge \neg g' \texttt{SAT}}{\langle g, \Sigma, \texttt{while}_{\texttt{k}}\ b\ \texttt{do}\ s, \rangle \Downarrow \langle g \wedge \neg g', \Sigma \rangle}\ \textit{big-whilefalse}$$

In this formulation, we assume that all loops in the program are syntactically annotated with a limit k (this may be a program-wide constant or be user-defined per loop). When $k > 0$, the rule *big-whiletrue* first checks if the loop condition is feasible, and if so it analyzes the loop body and recursively subsequent iterations of the loop with a limit $k - 1$. Each time the loop is analyzed, the rule *big-whilefalse* considers the case where the loop condition may be `false`, and if this is feasible, simply propagates the symbolic environment and resulting path condition. Note that when $k = 0$, we do not consider the case that the loop body executes further. This form of k-limited symbolic execution is unsound for verification and incomplete for bug finding.

In SSE, when we *do* have loop invariants[1], we can incorporate them into the verification condition no more than twice: once, the first time the loop invariant is encountered, and again for an "arbitrary" loop iteration. For this arbitrary iteration, we determine which variables could possibly be modified on the path back to the invariant through the loop (how might we do this?). Then, we quantify over a new set of symbolic values for all of those variables (setting them to "arbitrary" values).[2] Static symbolic execution can then proceed through the while loop to the rest of the program.

13.2 Symbolic Execution as a Generalization of Testing

Symbolic execution[3] is fundamentally a way to generalize testing. A test involves executing a program concretely on one specific input, and checking the results. In contrast, symbolic

[1]Note that some approaches or languages for verification support arbitrary invariant annotation to help prune paths or render the verification problem simpler; the approach is the same.

[2]This approach can be similarly applied to extend symbolic execution to function calls.

[3]The general term "symbolic execution" will refer to DSE hereon and whenever used without qualification.

execution considers how the program executes abstractly on a family of related inputs.

13.2.1 Illustration

Consider the following code example, where a, b, and c are user-provided inputs:

```
1  int x=0, y=0, z=0;
2  if(a) {
3    x = -2;
4  }
5  if (b < 5) {
6    if (!a && c) { y = 1; }
7    z = 2;
8  }
9  assert(x + y + z != 3);
```

Question: *What is an example input that will lead this assertion to fail? What path is it associated with?*

If we are good (or lucky) testers, we can stumble upon a combination of inputs that triggers the assertion to fail, and then generalize to the combination of input spaces that will lead to it (and hopefully fix it!).

Symbolic execution effectively inverts this process by describing the paths through the program symbolically. Instead of executing the code on concrete inputs (like a = 1, b = 2, and c = 1), it instead tracks execution in terms of symbolic inputs $a = \alpha, b = \beta, c = \gamma$. If a branch condition ever depends on unknown symbolic values, the symbolic execution engine simply chooses one branch to take, recording the condition on the symbolic values that would lead to that branch. We can *split* the state, and use a worklist algorithm to make sure we come back to the other branch.

For example, consider abstractly executing a path through the program above, keeping track of the (potentially symbolic) values of variables, and the conditions that must be true in order for us to take that path. We can write this in tabular form, showing the values of the path condition g and symbolic environment E after each line:

line	g	E
0	true	$a \mapsto \alpha, b \mapsto \beta, c \mapsto \gamma$
1	true	$\ldots, x \mapsto 0, y \mapsto 0, z \mapsto 0$
2	$\neg\alpha$	$\ldots, x \mapsto 0, y \mapsto 0, z \mapsto 0$
5	$\neg\alpha \wedge \beta \geqslant 5$	$\ldots, x \mapsto 0, y \mapsto 0, z \mapsto 0$
9	$\neg\alpha \wedge \beta \geqslant 5 \wedge 0 + 0 + 0 \neq 3$	$\ldots, x \mapsto 0, y \mapsto 0, z \mapsto 0$

In the example, we arbitrarily picked the path where the abstract value of a, i.e. α, is false, and the abstract value of b, i.e. β, is not less than 5. We build up a path condition out of these boolean predicates as we hit each branch in the code. The assignment to x, y, and z updates the symbolic state E with expressions for each variable; in this case we know they are all equal to 0. At line 9, we treat the assert statement like a branch. In this case, the branch expression evaluates to $0 + 0 + 0 \neq 3$ which is true, so the assertion is not violated.

Now, we can run symbolic execution again along another path (this style of analysis should feel familiar; note that I haven't given you termination guarantees, however, unlike in abstract interpretation! That's where the slacker vs. non-slacker debate comes back in...). We can do this multiple times, until we explore all paths in the program (*exercise to the reader: how many paths are there in the program above?*) or we run out of time. If we continue doing this, eventually we will explore the following path:

line	g	E
0	true	$a \mapsto \alpha, b \mapsto \beta, c \mapsto \gamma$
1	true	$\ldots, x \mapsto 0, y \mapsto 0, z \mapsto 0$
2	$\neg\alpha$	$\ldots, x \mapsto 0, y \mapsto 0, z \mapsto 0$
5	$\neg\alpha \wedge \beta < 5$	$\ldots, x \mapsto 0, y \mapsto 0, z \mapsto 0$
6	$\neg\alpha \wedge \beta < 5 \wedge \gamma$	$\ldots, x \mapsto 0, y \mapsto 1, z \mapsto 0$
6	$\neg\alpha \wedge \beta < 5 \wedge \gamma$	$\ldots, x \mapsto 0, y \mapsto 1, z \mapsto 2$
9	$\neg\alpha \wedge \beta < 5 \wedge \neg(0 + 1 + 2 \neq 3)$	$\ldots, x \mapsto 0, y \mapsto 1, z \mapsto 2$

Along this path, we have $\neg\alpha \wedge \beta < 5$. This means we assign y to 1 and z to 2, meaning that the assertion $0 + 1 + 2 \neq 3$ on line 9 is false. Symbolic execution has found an error in the program!

13.2.2 Symbolic Execution History and Industrial Use

Symbolic execution was originally proposed (as a way to generalize testing) in the 1970s [5, 18], but it relied on automated theorem proving, and the algorithms and hardware of that period weren't ready for widespread use. With recent advances in SAT/SMT solving and four decades of Moore's Law applied to hardware, symbolic execution is now practical in many more situations, and is used extensively in program analysis research as well as some emerging industry tools. One of the most prominent examples is the use of the PREfix to find errors in C/C++ code within Microsoft; the Klee symbolic execution engine is well known in open source and research contexts.

Of course, programs with loops have infinite numbers of paths, so exhaustive symbolic execution is not possible. Instead, tools take heuristics, as discussed above. To avoid analyzing complex library code, symbolic executors may use an abstract model of libraries. So, in its most common practical formulations, which uses heuristics scalability and termination, symbolic execution is typically less general than abstract interpretation. However, symbolic execution can often avoid approximating in places where AI must approximate to ensure termination. This means that symbolic execution can avoid giving false warnings; any error found by symbolic execution represents a real, feasible path through the program, and (as we will see) can be witnessed with a test case that illustrates the error.

13.3 Optional: Heap Manipulating Programs

We can extend the idea of symbolic execution to heap-manipulating programs. Consider the following extensions to the grammar of arithmetic expressions and statements, supporting memory allocation with *malloc* as well as dereferences and stores:

$$\begin{array}{rcl} a & ::= & \ldots \mid *a \mid malloc \\ S & ::= & \ldots \mid *a := a \end{array}$$

Now we can define memories as a basic memory μ that can be extended based on stores into the heap. The memory is modeled as an array, which allows SMT solvers to reason about it using the theory of arrays:

$$m \quad ::= \quad \mu \mid m[a_s \mapsto a_s]$$

Finally, we extend symbolic expressions to include heap reads:

$$a_s \quad ::= \quad \ldots \mid m[a_s]$$

Now we can define extended version of the arithmetic expression and statement execution semantics that take (and produce, in the case of statements) a memory:

$$\frac{\alpha \notin \Sigma, m}{\langle \Sigma, malloc, m \rangle \Downarrow \alpha} \textit{ big-malloc}$$

$$\frac{\langle \Sigma, a, m \rangle \Downarrow a_s}{\langle \Sigma, *a, m \rangle \Downarrow m[a_s]} \textit{ big-deref}$$

$$\frac{\langle \Sigma, a, m \rangle \Downarrow a_s \quad \langle \Sigma, a', m \rangle \Downarrow a'_s}{\langle g, \Sigma, m, *a := a' \rangle \Downarrow \langle g, \Sigma, m[a_s \mapsto a'_s] \rangle} \textit{ big-store}$$

Chapter 14

Concolic Testing

14.1 Introduction

We have discussed symbolic execution from two perspectives: as a method for forward verification condition generation, as well as a method that generalizes testing. We will continue to focus on this latter perspective by discussing key approaches that have allowed symbolic execution to find real bugs in practice.

14.1.1 Motivation

Companies today spend a huge amount of time and energy testing software to determine whether it does the right thing, and to find and then eliminate bugs. A major challenge is writing adequate test cases that cover all of the source code, as well as finding inputs that lead to difficult-to-trigger corner case defects.

Symbolic execution is a promising approach to exploring different execution paths through programs. However, it has significant limitations. For paths that are long and involve many conditions, SMT solvers may not be able to find satisfying assignments to variables that lead to a test case that follows that path. Other paths may be short but involve computations that are outside the capabilities of the solver, such as non-linear arithmetic or cryptographic functions. For example, consider the following function:

```
1  testme(int x, int y){
2      if(bbox(x)==y){
3          ERROR;
4      } else {
5          // OK
6      }
7  }
```

If we assume that the implementation of `bbox` is unavailable, or is too complicated for a theorem prover to reason about, then symbolic execution may not be able to determine whether the error is reachable.

14.1.2 Statically modeling functions

We have several options for symbolically executing a program with functions (like the one we developed for interprocedural dataflow analysis). Inlining is somewhat more practical here as we are not computing fixpoints. We can also simply symbolically execute the called methods, too; because we are not joining abstract state over multiple possible paths, we do not immediately lose precision as we would in interprocedural abstract interpretation.

If we continue to operate in a language with pre and postconditions specified at the function level (as we assumed in Hoare-Style verification), we can also use those to model function

behavior statically. Assuming pre- and post-conditions encoded in the same expression language as guards, $e_{\texttt{pre}}$ and $e_{\texttt{post}}$:

$$\frac{\langle e_{\texttt{post}}, \Sigma\rangle \Downarrow a_{\texttt{post}}}{\langle g, \Sigma, \texttt{return}\rangle \Downarrow \langle a_{\texttt{post}}, \Sigma\rangle} \textit{ big-return}$$

Question: what about function calls? Note that if the language involves heap-manipulation, this question becomes more or less difficult!

At some point, however, symbolic execution will reach the "edges" of the application: a library, system, or assembly code call. For certain libraries, a simpler version is available (such as `libc` implemented for embedded systems). Other tools allow custom code models, such as the implementation of a ramdisk to model kernel fs code. This is of course very labor intensive. Even when this code can be pulled in and executed symbolically, there are times that the code is simply too complicated to be tractably reasoned about statically, such as if it involves non-linear arithmetic.

The challenges of fully statically symbolically executing all code directly motivate *concolic testing*. Concolic testing combines **conc**rete execution (i.e. testing) with symb**olic** execution.[1]

14.1.3 Goals

We will consider the specific goal of automatically unit testing programs to find assertion violations and run-time errors such as divide by zero. We can reduce these problems to input generation: given a statement s in program P, compute input i such that $P(i)$ executes s.[2] For example, if we have a statement `assert x > 5`, we can translate that into the code:

```
1  if (!(x > 5))
2      ERROR;
```

Now if line 2 is reachable, the assertion is violated. We can play a similar trick with run-time errors. For example, a statement involving division `x = 3 / i` can be placed under a guard:

```
1  if (i != 0)
2      x = 3 / i;
3  else
4      ERROR;
```

14.2 Concolic execution overview

In concolic execution, symbolic execution is used to solve for inputs that lead along a certain path. However, when a part of the path condition is infeasible for the SMT solver to handle, we substitute values from a test run of the program. In many cases, this allows us to make progress towards covering parts of the code that we could not reach through either symbolic execution or randomly generated tests.

Consider the `testme` example from the motivating section. Although symbolic analysis cannot solve for values of `x` and `y` that allow execution to reach the error, we can generate random test cases. These random test cases are unlikely to reach the error: for each `x` there is only one `y` that will work, and random input generation is unlikely to find it. However, concolic testing can use the concrete value of `x` and the result of running `bbox(x)` in order to solve for a matching `y` value. Running the code with the original `x` and the solution for `y` results in a test case that reaches the error.

In order to understand how concolic testing works in detail, consider a more realistic and more complete example:

[1] The word concolic is a portmanteau of **conc**rete and symb**olic**

[2] This formulation is due to Wolfram Schulte

```
1  int double (int v) {
2      return 2*v;
3  }
4
5  void bar(int x, int y) {
6      z = double (y);
7      if (z == x) {
8          if (x > y+10) {
9              ERROR;
10         }
11     }
12 }
```

We want to test the function `bar`. We start with random inputs such as $x = 22, y = 7$. We then run the test case and look at the path that is taken by execution: in this case, we compute $z = 14$ and skip the outer conditional. We then execute symbolically along this path. Given inputs $x = x_0, y = y_0$, we discover that at the end of execution $z = 2 * y_0$, and we come up with a path condition $2 * y_0 \neq x_0$.

In order to reach other statements in the program, the concolic execution engine picks a branch to reverse. In this case there is only one branch touched by the current execution path; this is the branch that produced the path condition above. We negate the path condition to get $2 * y_0 == x_0$ and ask the SMT solver to give us a satisfying solution.

Assume the SMT solver produces the solution $x_0 = 2, y_0 = 1$. We run the code with that input. This time the first branch is taken but the second one is not. Symbolic execution returns the same end result, but this time produces a path condition $2 * y_0 == x_0 \wedge x_0 \leqslant y_0 + 10$.

Now to explore a different path we could reverse either test, but we've already explored the path that involves negating the first condition. So in order to explore new code, the concolic execution engine negates the condition from the second `if` statement, leaving the first as-is. We hand the formula $2 * y_0 == x_0 \wedge x_0 > y_0 + 10$ to an SMT solver, which produces a solution $x_0 = 30, y_0 = 15$. This input leads to the error.

The example above involves no problematic SMT formulas, so regular symbolic execution would suffice. The following example illustrates a variant of the example in which concolic execution is essential:

```
1  int foo(int v) {
2      return v*v%50;
3  }
4
5  void baz(int x, int y) {
6      z = foo(y);
7      if (z == x) {
8          if (x > y+10) {
9              ERROR;
10         }
11     }
12 }
```

Although the code to be tested in `baz` is almost the same as `bar` above, the problem is more difficult because of the non-linear arithmetic and the modulus operator in `foo`. If we take the same two initial inputs, $x = 22, y = 7$, symbolic execution gives us the formula $z = (y_0 * y_0)\%50$, and the path condition is $x_0 \neq (y_0 * y_0)\%50$. This formula is not linear in the input y_0, and so it may defeat the SMT solver.

We can address the issue by treating `foo`, the function that includes nonlinear computation, concretely instead of symbolically. In the symbolic state we now get $z = foo(y_0)$, and for $y_0 = 7$ we have $z = 49$. The path condition becaomse $foo(y_0) \neq x_0$, and when we negate this we get $foo(y_0) == x_0$, or $49 == x_0$. This is trivially solvable with $x_0 == 49$. We leave $y_0 = 7$ as before; this is the best choice because y_0 is an input to $foo(y_0)$ so if we change it, then

setting $x_0 = 49$ may not lead to taking the first conditional. In this case, the new test case of $x = 49, y = 7$ finds the error.

14.3 Implementation

Ball and Daniel [3] give the following pseudocode for concolic execution (which they call dynamic symbolic execution):

```
1  i = an input to program P
2  while defined(i):
3     p = path covered by execution P(i)
4     cond = pathCondition(p)
5     s = SMT(Not(cond))
6     i = s.model()
```

Broadly, this just systematizes the approach illustrated in the previous section. However, a number of details are worth noting:

First, when negating the path condition, there is a choice about how to do it. As discussed above, the usual approach is to put the path conditions in the order in which they were generated by symbolic execution. The concolic execution engine may target a particular region of code for execution. It finds the first branch for which the path to that region diverges from the current test case. The path conditions are left unchanged up to this branch, but the condition for this branch is negated. Any conditions beyond the branch under consideration are simply omitted. With this approach, the solution provided by the SMT solver will result in execution reaching the branch and then taking it in the opposite direction, leading execution closer to the targeted region of code.

Second, when generating the path condition, the concolic execution engine may choose to replace some expressions with constants taken from the run of the test case, rather than treating those expressions symbolically. These expressions can be chosen for one of several reasons. First, we may choose formulas that are difficult to invert, such as non-linear arithmetic or cryptographic hash functions. Second, we may choose code that is highly complex, leading to formulas that are too large to solve efficiently. Third, we may decide that some code is not important to test, such as low-level libraries that the code we are writing depends on. While sometimes these libraries could be analyzable, when they add no value to the testing process, they simply make the formulas harder to solve than they are when the libraries are analyzed using concrete data.

14.4 Concolic Path Condition Soundness

Concolic execution is motivated by the presence of subexpressions within a path condition that are difficult for a SMT solver to reason about. The key idea of concolic execution is to replace these subexpressions with appropriate concrete values. Where possible, we would like this replacement to be *sound*. Intuitively, a replacement is sound if any solution to the new path condition is also a solution to the old one. This means that even after the substitution, concolic execution will successfully drive the program down the desired path. Let's make this idea more formal.

Let g be a negated path condition. Let M be a map from symbolic constants α to integers n. We write $[M]g$ for the boolean expression we get by substituting all the symbolic constants in g with the corresponding integer values given in M; this is only defined if the free symbolic constants $FC(g)$ are the same as $\mathit{domain}(M)$. We define $[M]a_s$ similarly for substitution of symbolic constants with values in arithmetic expressions.

Given g and a map M that represents the inputs to a concrete test case execution, concolic execution may replace a subexpression a_s of g with the concrete value n achieved in testing.

Note that $n = [M]a_s$. Let the new guard be $g' = [n/a_s]g$ (again, we consider this *after* negating the last constraint in the path).

We say that g' is a *sound* concolic path condition if for all alternative test inputs M' such that $[M']g'$ is true, we have $[extend(M', M)]g$ true. Here, the $extend$ function extends the symbolic constants in M' with any that are necessary to match the domain of M. More precisely, $\forall \alpha' \in domain(M'), extend(M', M)[\alpha'] = M'[\alpha']$ and $\forall \alpha \in (domain(M) - domain(M')), extend(M', M)[\alpha] = M[\alpha]$.

In class we saw an example of a path condition g and a sound concolic replacement g' for it. In particular, g was $x_0 == (y_0 * y_0)\%50$ after negation and g' was $x_0 == 49$ after negation. This is trivially sound because the only solution is $x_0 == 49$, which when extended with $y_0 == 7$ from the original test case yields a new test input that fulfills the original path condition $x_0 == (y_0 * y_0)\%50$.

As an exercise:

- Give an example path condition g, test input M, and concolic path condition g' resulting from replacing a subexpression a_s of g with a concrete value $n = [M]a_s$, such that g' is *unsound*.
- Witness the unsoundness by also providing a test input M' that satisfies g' but not g.
- Give a condition on g, M, g' and/or a_s that is sufficient to ensure that g' is sound.
- Prove that your condition is sufficient for soundness.

14.5 Acknowledgments

The structure of these notes and the examples are adapted from a presentation by Koushik Sen.

Chapter 15

Program Synthesis

Note: A complete, if lengthy, resource on inductive program synthesis is the book "Program Synthesis" by Gulwani *et. al* [12]. You need not read the whole thing; I encourage you to investigate the portions of interest to you, and skim as appropriate. Many references in this document are drawn from there; if you are interested, it contains many more.

15.1 Program Synthesis Overview

The problem of program synthesis can be expressed as follows:

$$\exists P \,.\, \forall x \,.\, \varphi(x, P(x))$$

In the setting of *constructive* logic, proving the validity of a formula that begins with an existential involves coming up with a *witness*, or concrete example, that can be plugged into the rest of the formula to demonstrate that it is true. In this case, the witness is a program P that satisfies some specification φ on all inputs. We take a liberal view of P in discussing synthesis, as a wide variety of artifact types have beeen successfully synthesized (anything that reads inputs or produces outputs). Beyond (relatively small) program snippets of the expected variety, this includes protocols, interpreters, classifiers, compression algorithms or implementations, scheduling policies, and cache coherence protocols for multicore processors. The specification φ is an expression of the user intent, and may be expressed in one of several ways: a formula, a reference implementation, input/output pairs, traces, demonstrations, or a syntactic *sketch*, among other options.

Program synthesis can thus be considered along three dimensions:

(1) Expressing user intent. User intent (or φ in the above) can be expressed in a number of ways, including logical specifications, input/output examples [8] (often with some kind of user- or synthesizer-driven interaction), traces, natural language [6, 11, 19], or full- or partial programs [33]. In this latter category lies reference implementations, such as executable specifications (which give the desired output for a given input) or declarative specifications (which check whether a given input/output pair is correct). Some synthesis techniques allow for multi-modal specifications, including pre- and post- conditions, safety assertions at arbitrary program points, or partial program templates.

Such specifications can constrain two aspects of the synthesis problem:

- **Observable behavior**, such as an input/output relation, a full executable specification or safety property. This specifies *what* a program should compute.
- **Structural properties**, or internal computation steps. These are often expressed as a sketch or template, but can be further constrained by assertions over the number or variety of operations in a synthesized programs (or number of iterations, number of cache

misses, etc, depending on the synthesis problem in question). Indeed, one of the key principles behind the scaling of many modern synthesis techniques lie in the way they syntactically restrict the space of possible programs, often via a sketch, grammar, or DSL.

.

Note that basically all of the above types of specifications can be translated to constraints in some form or another. Techniques that operate over multiple types of specifications can overcome various challenges that come up over the course of an arbitrary synthesis problem. Different specification types are more suitable for some types of problems than others. In addition, trace- or sketch-based specifications can allow a synthesizer to decompose a synthesis problems into intermediate program points.

Question: how many ways can we specify a sorting algorithm?

(2) Search space of possible programs. The search space naturally includes programs, often constructed of subsets of normal programming languages. This can include a predefined set of considered operators or control structures, defined as grammars. However, other spaces are considered for various synthesis problems, like logics of various kinds, which can be useful for, e.g., synthesizing graph/tree algorithms.

(3) Search technique. At a high level, there are two general approaches to logical synthesis:

- Deductive (or classic) synthesis (e.g., [22]), which maps a high-level (e.g. logical) specification to an executable implementation, classically using a theorem prover. Such approaches are efficient and provably correct: thanks to the semantics-preserving rules, only correct programs are explored. However, they require complete specifications and sufficient axiomatization of the domain. These approaches are classically applied to e.g., controller synthesis.
- Inductive (sometimes called syntax-guided) synthesis, which takes a partial (and often multi-modal) specification and constructs a program that satisfies it. These techniques are more flexible in their specification requirements and require no axioms, but often at the cost of lower efficiency and weaker bounded guarantees on the optimality of synthesized code.

Deductive synthesis shares quite a bit in common, conceptually, with compilation: rewriting a specification according to various rules to achieve a new program in at a different level of representation. However, deductive synthesis approaches assume a complete formal specification of the desired user intent was provided. In many cases, this can be as complicated as writing the program itself.

This has motivated new inductive synthesis approaches, towards which considerable modern research energy has been dedicated. This category of techniques lends itself to a wide variety of search strategies, including brute-force or enumerative [1] (you might be surprised!), probabilistic inference/belief propagation [10], or genetic programming [17]. Alternatively, techniques based on logical reasoning delegate the search problem to a constraint solver. We will spend more time on this set of techniques.

15.2 Deductive Synthesis

We will very briefly overview Denali [16], a prototypical deductive synthesis technique for *superoptimization*.[1] Denali seeks to generate short sequences of provably optimal loop-free

[1] This explanation is further illustrated using the associated lecture slides.

machine instructions, for use primarily incompilation. While compilers generate reasonably good code, there are cases in which we would instead prefer provably optimal code. Generating such code is the task of a superoptimizer (so-called because the title of optimization "has been given to a field that does not aspire to optimize but only to improve"). Early approaches for superoptimization attempted to enumerate via brute force (in order of increasing length) efficient sequences of instructions, with correctness checked by hand and against a set of test cases. This correctness critierion is challenging to confirm, however, and does not necessarily result in optimality.

Joshi et al. propose an approach to superoptimization based on theorem proving. The "obvious" approach (which they do *not* take) would be to, given a desired program fragment P, express in formal logic "no program of the target architecture computes P in at most N cycles." However, this obvious approach is very difficult to manage with a theorem prover, because it must be expressed using nested quantifiers.

Instead, they propose a process based on the idea that for sufficiently simple programs, equivalence between a desired P and some alternative implementation M for all inputs is essentially the universal validity of an equality between two vectors of terms (the one M computes, and the terms specified by P in the computatoin). This type of equivalience can be proved by matching, which is a well understood technique in theorem proving.

To do this, their technique, named Denali, takes as input a program P written in a DSL for the associated target architecture. It then constructs an *E-graph* using the specified desired program P as input. An E-graph is a term DAG corresponding to the expression to be synthesized, augmented with an equivalence relation on the nodes of the DAG. Two nodes are equivalent if the terms they represent are identical in value. Denali then uses a theorem prover, along with two sets of axioms (encoding *instruction semantics*—an interpreter for the target language, effectively—and algebraic properties—memory modeling, mostly), to search the e-graph for the most efficient way to compute the expression.

15.3 Inductive Synthesis

Inductive synthesis uses inductive reasoning to construct programs in response to partial specifications. The program is synthesized via a symbolic interpretation of a space of candidates, rather than by deriving the candidate directly. So, to synthesize such a program, we basically only require an interpreter, rather than a sufficient set of derivation axioms. Inductive synthesis is applicable to a variety of problem types, such as string transformation (FlashFill) [9], data extraction/processing/wrangling [8, 32], layout transformation of tables or tree-shaped structures [34], graphics (constructing structured, repetitive drawings) [13, 4], program repair [23, 20] (spoiler alert!), superoptimization [16], and efficient synchronization, among others.

Inductive synthesis consists of several family of approaches; we will overview several prominent examples, without claiming to be complete.

15.3.1 SKETCH, CEGIS, and SyGuS

SKETCH is a well-known synthesis system that allows programs to provide partial programs (a sketch) that expresses the high-level structure of the intended implementation but leaves holes for low-level implementation details. The synthesizer fills these holes from a finite set of choices, using an approach now known as Counterexample-guided Inductive Synthesis (CEGIS) [33, 30]. This well-known synthesis architecture divies the problem into *search* and *verification* components, and uses the output from the latter to refine the specification given to the former.

We have a diagram to illustrate on slides.

Syntax-Guided Synthesis (or SyGuS) formalizes the problem of program synthesis where specification is supplemented with a syntactic template. This defines a search space of possible programs that the synthesizer effectively traverses. Many search strategies exist; two especially well-known strategies are *enumerative search* (which can be remarkably effective, though rarely scales), and *deductive* or *top down* search, which recursively reduces the problem into simpler sub-problems.

15.3.2 Oracle-guided synthesis

Templates or sketches are often helpful and easy to write. However, they are not always available. Beyond search or enumeration, constraint-based approaches translate a program's specification into a constraint system that is provided to a solver. This can be especially effective if combined with an outer CEGIS loop that provides oracles.

This kind of synthesis can be effective when the properties we care about are relatively easy to verify. For example, imagine we wanted to find a maximum number m in a list l.

Turn to the handout, which asks you to specify this as a synthesis problem...

$$\exists P_{max} \forall l, m : P_{max}(l) = m \Rightarrow (m \in l) \wedge (\forall x \in l : m \geqslant x)$$

Proving this involves the following formula:

$$\forall l, m : P_{max}(l) = m \Rightarrow (m \in l) \wedge (\forall x \in l : m \geqslant x)$$

Note that instead of proving that a program satisfies a given formula, we can instead disprove its negation, which is:

$$\exists l, m : (P_{max}(l) = m) \wedge (m \notin l \vee \exists x \in l : m < x)$$

If the above is satisfiable, a solver will give us a counterexample, which we can use to strengthen the specification–so that next time the synthesis engine will give us a program that excludes this counterexample. We can make this counterexample more useful by asking the solver not just to provide us with an input that produces an error, but also to provide the corresponding correct output m^*:

$$\exists l, m^* : (P_{max}(l) \neq m^*) \wedge (m^* \in l) \wedge (\forall x \in l : m^* \geqslant x)$$

This is a much stronger constraint than the original counterexample, as it says what the program should output in this case rather than one example of something it should not output. Thus we now have an additional test case for the next round of synthesis. This counterexample-guided sythesis approach was originally introduced for SKETCH, and was generalized to oracle-guided inductive synthesis by Jha and Seshia. Different oracles have been developed for this type of synthesis. We will discussed component-based oracle-guided program synthesis in detail, which illustrates the use of distinguishing oracles.

15.4 Oracle-guided Component-based Program Synthesis

Problem statement and intuition.[2] Given a set of input-output pairs $< \alpha_0, \beta_0 > \ldots < \alpha_n, \beta_n >$ and N components $f_1, \ldots f_n$, the goal is to synthesize a function f out of the components such that $\forall \alpha_i . f(\alpha_i)$ produces β_i. We achieve this by constructing and solving a set of constraints

[2]These notes are inspired by Section III.B of Nguyen *et al.*, ICSE 2013 [27] ...which provides a really beautifully clear exposition of the work that originally proposed this type of synthesis in Jha *et al.*, ICSE 2010 [15].

over f, passing those constraints to an SMT solver, and using a returned satisfying model to reconstruct f. In this approach, the synthesized function will have the following form:

0	$z_0 := \texttt{input}^0$
1	$z_1 := \texttt{input}^1$
...	...
m	$z_m := \texttt{input}^m$
$m+1$	$z_{m+1} := f_?(z_?, \ldots, z_?)$
$m+2$	$z_{m+2} := f_?(z_?, \ldots, z_?)$
...	...
$m+n$	$z_{m+n} := f_?(z_?, \ldots, z_?)$
$m+n+1$	`return` $z_?$

The thing we have to do is fill in the ? indexes in the program above. These indexes essentially define the order in which functions are invoked and what arguments they are invoked with. We will assume that each component is used once, without loss of generality, since we can duplicate the components.

Definitions. We will set up the problem for the solver using two sets of variables. One set represents the input values passed to each component, and the output value that component produces, when the program is run for a given test case. We use $\overrightarrow{\chi}_i$ to denote the vector of input values passed to component i and r_i to denote the result value computed by that component. So if we have a single component (numbered 1) that adds two numbers, the input values $\overrightarrow{\chi}_1$ might be (1,3) for a given test case and the output r_1 in that case would be 4. We use Q to denote the set of all variables representing inputs and R to denote the set of all variables representing outputs:

$$\begin{aligned} Q &:= \bigcup_{i=1}^{N} \overrightarrow{\chi}_i \\ R &:= \bigcup_{i=1}^{N} r_i \end{aligned}$$

We also define the overall program's inputs to be the vector $\overrightarrow{Y}$ and the program's output to be r.

The other set of variables determines the location of each component, as well as the locations at which each of its inputs were defined. We call these *location variables*. For each variable x, we define a location variable l_x, which denotes where x is defined. Thus l_{r_i} is the location variable for the result of component i and $\overrightarrow{l_{\chi_i}}$ is the vector of location variables for the inputs of component i. So if we have $l_{r_3} = 5$ and $\overrightarrow{l_{\chi 3}}$ is (2,4), then we will invoke component #3 at line 5, and we will pass variables z_2 and z_4 to it. L is the set of all location variables:

$$L := \{l_x | x \in Q \cup R \cup \overrightarrow{Y} \cup r\}$$

We will have two sets of constraints: one to ensure the program is *well-formed*, and the other that ensures the program encodes the desired *functionality*.

Well-formedness. ψ_{wfp} denotes the well-formedness constraint. Let $M = |\overrightarrow{Y}| + N$, where N is the number of available components:

$$\begin{aligned} \psi_{wfp}(L,Q,R) \stackrel{def}{=} & \bigwedge_{x \in Q} (0 \leqslant l_x < M) \wedge \bigwedge_{x \in R} (|\overrightarrow{Y}| \leqslant l_x < M) \wedge \\ & \psi_{cons}(L,R) \wedge \psi_{acyc}(L,Q,R) \end{aligned}$$

The first line of that definition says that input locations are in the range 0 to M, while component output locations are all defined after program inputs are declared. ψ_{cons} and ψ_{acyc}

dictate that there is only one component in each line and that the inputs of each component are defined before they are used, respectively:

$$\psi_{cons}(L,R) \stackrel{def}{=} \bigwedge_{x,y\in R, x\not\equiv y} (l_x \neq l_y)$$
$$\psi_{acyc}(L,Q,R) \stackrel{def}{=} \bigwedge_{i=1}^{N} \bigwedge_{x\in \overrightarrow{\chi}_i, y\equiv r_i} l_x < l_y$$

Functionality. ϕ_{func} denotes the functionality constraint that guarantees that the solution f satisfies the given input-output pairs:

$$\phi_{func}(L,\alpha,\beta) \stackrel{def}{=} \psi_{conn}(L,\overrightarrow{Y},r,Q,R) \wedge \phi_{lib}(Q,R) \wedge (\alpha = \overrightarrow{Y}) \wedge (\beta = r)$$
$$\psi_{conn}(L,\overrightarrow{Y},r,Q,R) \stackrel{def}{=} \bigwedge_{x,y\in Q\cup R\cup \overrightarrow{Y}\cup\{r\}} (l_x = l_y \Rightarrow x = y)$$
$$\phi_{lib}(Q,R) \stackrel{def}{=} (\bigwedge_{i=1}^{N} \phi_i(\overrightarrow{\chi}_i, r_i))$$

ψ_{conn} encodes the meaning of the location variables: If two locations are equal, then the values of the variables defined at those locations are also equal. ϕ_{lib} encodes the semantics of the provided basic components, with ϕ_i representing the specification of component f_i. The rest of ϕ_{func} encodes that if the input to the synthesized function is α, the output must be β.

Almost done! ϕ_{func} provides constraints over a single input-output pair α_i, β_i, we still need to generalize it over all n provided pairs $\{<\alpha_i,\beta_i> | 1 \leqslant i \leqslant n\}$:

$$\theta \stackrel{def}{=} (\bigwedge_{i=1}^{n} \phi_{func}(L,\alpha_i,\beta_i)) \wedge \psi_{wfp}(L,Q,R)$$

θ collects up all the previous constraints, and says that the synthesized function f should satisfy all input-output pairs and the function has to be well formed.

LVal2Prog. The only real unknowns in all of θ are the values for the location variables L. So, the solver that provides a satisfying assignment to θ is basically giving a valuation of L that we then turn into a constructed program as follows:

Given a valuation of L, `Lval2Prog(L)` converts it to a program as follows: The i^{th} line of the program is $z_i = f_j(z_{\sigma_1},...,r_{\sigma_\eta})$ when $l_{r_j} == i$ and $\bigwedge_{k=1}^{\eta} (l_{\chi_j^k} == \sigma_k)$, where η is the number of inputs for component f_j and χ_j^k denotes the k^{th} input parameter of component f_j. The program output is produced in line l_r.

Example. Assume we only have one component, +. + has two inputs: χ_+^1 and χ_+^2. The output variable is r_+. Further assume that the desired program f has one input Y_0 (which we call $\texttt{input}^0$ in the actual program text) and one output r. Given a mapping for location variables of: $\{l_{r_+} \mapsto 1, l_{\chi_+^1} \mapsto 0, l_{\chi_+^2} \mapsto 0, l_r \mapsto 1, l_Y \mapsto 0\}$, then the program looks like:

$$\begin{array}{ll} 0 & z_0 := \texttt{input}^0 \\ 1 & z_1 := z_0 + z_0 \\ 2 & \texttt{return}\ z_1 \end{array}$$

This occurs because the location of the variables used as input to + are both on the same line (0), which is also the same line as the input to the program (0). l_r, the return variable of the program, is defined on line 1, which is also where the output of the + component is located. (l_{r_+}). We added the return on line 2 as syntactic sugar.

Chapter 16

Fuzz Testing

So far, we have looked at program analysis techniques that perform deep introspection of the source code in order to over-approximate or under-approximate program behavior, often with the goal of finding program bugs. But what if we did not want to look at the program source code at all? This might be the case either if we did not have access to source code (e.g. when analyzing third-party binaries) or because we are dealing with programs so large and complex that any sort of static analysis with super-linear complexity is too expensive. This chapter provides an introduction to a technique that is very popular for stress testing industrial code. This chapter was largely adapted from Rohan Padhye's Ph.D. thesis [28].

16.1 Random Fuzzing

Practioners have long known that simply generating test inputs at *random* is a scalable and surprisingly effective method for finding implementation faults in computer systems. Random test generation was first popularized for finding faults in hardware in the 1970s and 80s: random test-input generators were developed for sequential circuits, memories, ICs, floating point units, cache controllers, etc.

Random test-case generation as a methodology for finding software bugs was initially dismissed: Myers' 1979 book *The Art of Software Testing* [26] states "the least effective methodology of all is random-input testing". However, by the 1980s random testing was found to be "more cost effective" than systematic techniques and "a useful validation tool" that achieves "a very high degree of coverage" [14]. Many of these results reflect experiences in testing software that operated on a fixed set of numeric inputs, such as computer simulations.

In 1990, Miller et al. [24] developed `fuzz`, a tool for testing the reliability of Unix utilities by generating random sequences of characters as input[1]. They were able to crash dozens of standard widely used Unix utilities including `vi`, `emacs`, `as`, `ftp`, `spell`, and `uniq` by simply feeding random input data generated by `fuzz`. A common cause of these crashes was segfaults; many of the tested programs had input-validation bugs such as missing size checks or improper format strings that could cause the programs to read/write memory out of bounds when presented with unexpected inputs. Such *buffer overflow* bugs were and remain serious security vulnerabilities[2].

Today, *fuzz testing*, or simply *fuzzing*, refers to any test-input generation technique that produces inputs using some randomized algorithm. The input generator is itself sometimes

[1] Apparently, one of the authors accidentally discovered fuzz testing when working from home one "dark and stormy night"; the rain introduced noise in the phone lines which were transmitting his commands to a remote Unix system and caused programs at the other end to crash [24].

[2] MITRE Corporatation's Common Weakness Enumeration (CWE) list ranks buffer overflows as number 1 in the top 25 most dangerous software errors in 2019 [25].

referred to as a *fuzzer*. In the three decades following Miller et al.'s work, fuzz testing has become a rich field of research for finding security vulnerabilities.

The key advantage of fuzz testing over systematic techniques such as symbolic execution is scalability: a randomized search can explore many program behaviors quickly and can be easily parallelized. Possibly fueled by the increasing availability of cheap computing resources, fuzz testing has become one of the predominant automated testing methods used in practice. For example, Google's ClusterFuzz system has found more than 16,000 bugs in the Chrome web browser and over 11,000 bugs across 160+ open-source projects by January 2019 [7].

Modern fuzzers rarely generate inputs randomly from scratch: it is very unlikely that inputs constructed as purely random sequences of bytes will exercise a non-trivial fraction of a complex software system. The two broad approaches to smarter input generation include *model-based fuzzing* and *mutation-based fuzzing*.

Model-based fuzzers generate inputs based on some understanding of what kind of inputs a program expects. Although this might seem unintuitive—the goal of fuzzing is to generate *unexpected* inputs that reveal software bugs—the idea is that generating inputs having some basic structure or syntax will guarantee that certain parts of a test program's code logic are exercised. For example, grammar-based fuzzing techniques use context-free grammar specifications to generate strings belong to a particular language.

Mutation-based fuzzers generate inputs by performing random changes on valid *seed* inputs. The idea is that making small random changes to a valid input, such as flipping some bits in an input to a program that processes binary data (e.g. a media player), or inserting random keywords in a text input to a parser (e.g. in a database query processor) will correspond to subtle changes in the execution path of the test program through its control-flow graph. Random mutations will create new, previously unseen and possibly unexpected inputs, while retaining much of the syntax, structure, and other features of the valid seed input. This idea has been used extensively by security-oriented fuzzers for discovering memory-corrution errors in commonly used software such as Unix utilities, network protocol implementations, and C libraries.

Both model-based fuzzers and mutation-based fuzzers make use of some knowledge about what kind of inputs a program expects. Both techniques as described above are *black box*; that is, they do not analyze the test program's source code or collect any additional information during program execution. Such black box testing is incredibly efficient, especially when compared to *white box* techniques such as symbolic execution which need to collect path constraints for every execution. Black-box fuzzers are also embarassingly parallelizable since the generation of every input is independent from every other input. However, a direct consequence of this fact is that the probability of generating an input that reveals a bug is the exact same when generating the very first input as it is when generating say the hundred millionth input. Some of the most important questions about the random testing strategy include *"'How long should it run?"* and *"Has it covered all the important cases?"*.

One way to measure the quality of a set of test inputs generated by a fuzzer is to use proxy metrics such as *code coverage*, which correspond to the amount or fraction of program code that is exercised by test inputs. Common granularities of code coverage include *line coverage*, *statement coverage*, *branch coverage*, *basic-block coverage*, and *edge coverage* (the latter two refer to nodes and edges in a program's control-flow graph respectively). A straightforward strategy for tracking the progress of a fuzzing session is to measure the code coverage achieved by all the inputs generated so far; fuzzing is no longer viable when the rate of increasing code coverage falls below a certain threshold. However, measuring code coverage requires test programs to be *instrumented* with code that tracks which parts of the program are being exercised when executing test inputs. This instrumentation adds performance overhead, and can reduce the overall fuzzing efficiency.

Collecting code coverage during a fuzzing session does brings one very important advantage, at least to mutation-based fuzzers. The coverage information can be used to augment

Algorithm 1 The coverage-guided fuzzing algorithm

Input: an instrumented test program p, a set of initial seed inputs $\mathcal{I}$
Output: a corpus of automatically generated inputs $\mathcal{S}$, a set of failing test inputs $\mathcal{F}$

```
1:  S ← I
2:  F ← ∅
3:  totalCoverage ← ∅
4:  repeat                                                    ▷ Main fuzzing loop
5:      for i in S do
6:          if sample FUZZPROB(i) then
7:              i' ← MUTATE(i)                                ▷ Generate new test input i'
8:              coverage, result ← EXECUTE(p, i')             ▷ Run test with new input i'
9:              if result = FAILURE then
10:                 F ← F ∪ {i'}
11:             else if coverage ⊈ totalCoverage then
12:                 S ← S ∪ {i'}                              ▷ Save i' if new code coverage achieved
13:                 totalCoverage ← totalCoverage ∪ coverage
14:             end if
15:         end if
16:     end for
17: until given time budget expires
18: return S, F
```

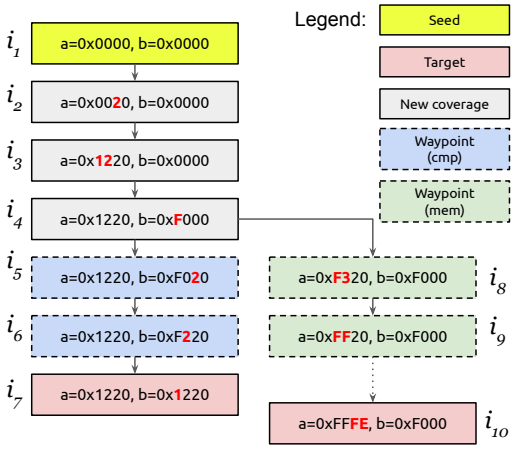
the set of *seed* inputs: automatically generated (i.e., fuzzed) inputs that exercise previously uncovered code can be used as the basis for subsequent mutation. In this way, fuzzing can become *feedback-directed* and test inputs can *evolve* over time. The vast majority of recent progress in fuzz testing, both in terms of new research and new discoveries of serious software bugs, has stemmed from the field of *coverage-guided fuzzing* (CGF).

16.2 Coverage-Guided Fuzzing (CGF)

Algorithm 1 describes how CGF works at a high level. The CGF algorithm takes as input an instrumented test program p and a set of user-provided *seed inputs* $\mathcal{I}$. CGF maintains three global states: (1) $\mathcal{S}$ is a set of saved inputs to be mutated by the algorithm, (2) $\mathcal{F}$ is a set of bug-revealing inputs corresponding to test failures, and (2) $totalCoverage$ tracks the cumulative coverage of the program on the inputs in $\mathcal{S}$. CGF can track any kind of coverage; in practice, branch coverage or edge coverage is commonly used. $\mathcal{S}$ is initialized to the set of user-provided seed inputs (Line 1) and $totalCoverage$ is initialized to the empty set (Line 3). The main fuzzing loop of CGF (Line 4) keeps making passes over the set of inputs (Line 5), selecting an input i from the set $\mathcal{S}$. With some probability (Line 6) determined by an implementation-specific heuristic function FUZZPROB(i), CGF decides whether to mutate the input i or not. If i is selected for mutation, CGF randomly mutates i to generate i' (Line 7). The random mutation can be selected from a set of predefined mutations such as bit flipping, byte flipping, arithmetic increment and decrement of integer values, replacing of bytes with handpicked interesting values, etc. CGF then executes the program p with the newly generated input i' (Line 8). The coverage corresponding to this execution is collected into the variable $coverage$. The variable $result$ whether the execution terminated normally or abnormally (e.g. with a crash or time-out). Inputs corresponding to test failures are added to a set $\mathcal{F}$ (Line 10). If the observed coverage $coverage$ when executing a non-failing input contains some new coverage point that is not present in the global cumulative coverage $totalCoverage$ (Line 11), then the new input i' is added to the set of saved inputs $\mathcal{S}$ (Line 12) and $totalCoverage$ is updated to include the new

```
1  void* Test(int16_t a, int16_t b) {
2    if (a % 3 == 2) {
3      if (a > 0x1000) {
4        if (b >= 0x0123) {
5          if (a == b) {
6            abort();
7          } else {
8            return malloc(a);
9          }
10       }
11     }
12   }
13 }
```

(a) Sample function in the test program. Parameters `a` and `b` are the test inputs.



(b) Sample fuzzed inputs starting with initial seed $a = 0$, $b = 0$. Arrows indicate mutations.

Figure 16.1: A motivating example for custom fuzzing waypoints.

coverage (Line 13). The input i' will then get mutated during a future iteration of the fuzzing loop. The fuzzing loop continues until a time budget has expired (Line 17). Finally, the generated test corpus $\mathcal{S}$ and the set of failing inputs $\mathcal{F}$ are returned as the result of fuzzing (Line 18).

16.2.1 Contemporary CGF Tools: AFL and libFuzzer

CGF was popularized by AFL [35]—which stands for *American Fuzzy Lop*—an open-source fuzzing tool developed by Michał Zalewski at Google. AFL starts fuzzing using a user-provided set of seed input files, corresponding to set $\mathcal{I}$ in Algorithm 1. The mutations applied by AFL to generate new inputs include:

- Bitflips/byteflips at random locations.
- Setting bytes to random or interesting (0, `MAX_INT`) values at random locations.
- Deleting/cloning random blocks of bytes.

AFL also occasionally performs *splicing* mutations, more commonly called a *crossover* mutation. For a candidate input i, a splicing mutation chooses a random input i' in $\mathcal{S}$ and pastes a random sub-sequence from i' at a random offset in i. This stage runs only when AFL has not discovered new coverage in several cycles of the main fuzzing loop. AFL also allows users to specify a *dictionary* of keywords or *magic* byte sequenees that are then randomly inserted into mutated inputs.

LibFuzzer is another widely used CGF tool that targets the LLVM platform. Since around 2016, libFuzzer [21] has been included as part of the LLVM project. Together, AFL and libFuzzer have been used to discover thousands of security vulnerabilities, mostly in C/C++ programs such as Google Chrome, OpenSSL, Mozilla Firefox, Adobe Flash, VLC Media Player, and others.

16.3 Domain-Specific Fuzzing with Waypoints

Consider the sample test program in Figure 16.1a. The function `Test` takes as input two 16-bit integers, `a` and `b`. A common test objective is to generate inputs that maximize code coverage in this program. We apply Algorithm 1 to perform CGF on this test program. Let us assume that we start with the *seed input*: a=0x0000, b=0x0000. The seed input does not satisfy

the condition at Line 2. The CGF algorithm randomly mutates this seed input and executes the test program on the mutated inputs while looking for new code coverage. Figure 16.1b depicts in grey boxes a series of sample inputs which may be saved by CGF, starting with the initial seed input i_1 in an yellow box. A solid arrow between two inputs, say i and i', indicates that the input i is mutated to generate i'. After some attempts, CGF may mutate the value of `a` in i_1 to a value such as 0x0020, which satisfies the condition at Line 2. Since such an input leads to new code being executed, it gets saved to $\mathcal{S}$. In Fig. 16.1b, this is input i_2. Small, byte-level mutations enable CGF to subsequently generate inputs that satisfy the branch condition at Line 3 and Line 4 of Fig. 16.1a. This is because there are many possible solutions that satisfy the comparisons `a > 0x1000` and `b >= 0x0123` respectively; we call these *soft* comparisons. Fig. 16.1b shows the corresponding inputs in our example: i_3 and i_4. However, it is much more difficult for CGF to generate inputs to satisfy comparisons such as `a == b` at Line 5; we call these *hard* comparisons. Random byte-level mutations on inputs i_1–i_4 are unlikely to produce an input where `a == b`. Therefore, the code at Line 6 may not be exercised in a reasonable amount of time using conventional CGF.

Now, consider another test objective, where we would like to generate inputs that maximize the amount of memory that is dynamically allocated via `malloc`. This objective is useful for generating stress tests or to discover potential out-of-memory related bugs. The CGF algorithm enables us to generate inputs that invoke `malloc` statement at Line 8, such as i_4. However, this input only allocates `0x1220` bytes (i.e., just over 4KB) of memory. Although random mutations on this input are likely to generate inputs that allocate larger amount of memory, CGF will never save these because they have the same coverage as i_4. Thus, it is unlikely that CGF will discover the *maximum* memory-allocating input in a reasonable amount of time.

Both of the challenges listed above can be addressed if we save some useful intermediate inputs to $\mathcal{S}$ regardless of whether they increase code coverage. Then, random mutations on these intermediate inputs may produce inputs achieving our test objectives. Such intermediate inputs are called *waypoints*. For example, to overcome hard comparisons such as `a == b`, we want to save intermediate inputs if they maximize the number of common bits between `a` and `b`. Let us call this strategy `cmp`. The blue boxes in Fig. 16.1b show inputs that may be saved to $\mathcal{S}$ when using the `cmp` strategy for waypoints. In such a strategy, the inputs i_5 and i_6 are saved to $\mathcal{S}$ even though they do not achieve new code coverage. Now, input i_6 can easily be mutated to input i_7, which satisfies the condition `a == b`. Thus, we easily discover an input that triggers `abort` at Line 6 of Fig. 16.1a. Similarly, to achieve the objective of maximizing memory allocation, we save waypoints that allocate more memory at a given call to `malloc` than any other input in $\mathcal{S}$. Fig. 16.1b shows sample waypoints i_8 and i_9 that may be saved with this strategy, called `mem`. The dotted arrow from i_9 to i_{10} indicates that, after several such waypoints, random mutations will eventually lead us to generating input i_{10}. This input causes the test program to allocate the maximum possible memory at Line 8, which is almost 64KB.

Now, consider a change to the condition at Line 4 of Figure 16.1a. Instead of an inequality, suppose the condition is `b == 0x0123`. To generate inputs that invoke `malloc` at Line 8, we first need to overcome a hard comparison of `b` with `0x0123`. We can combine the two strategies for saving waypoints as follows: save a new input i if *either* it increases the number of common bits between operands of hard comparisons *or* if it increases the amount of memory allocated at some call to `malloc`. A combination of these strategies was demonstrated [29] to automatically generate PNG bombs and LZ4 bombs, i.e. tiny inputs that allocate 2–4 GB of memory, when fuzzing `libpng` and `libarchive` respectively.

The general idea of changing the input-saving criteria beyond code coverage by using domain-specific feedback has been used to generate worst-case performance bugs, exercise state machines in data-intensive applications, discover side-channel leakages in privacy-sensitive applications, and even synthesize input sequeneces to play video games.

Bibliography

[1] R. Alur, R. Bodík, E. Dallal, D. Fisman, P. Garg, G. Juniwal, H. Kress-Gazit, P. Madhusudan, M. M. K. Martin, M. Raghothaman, S. Saha, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In M. Irlbeck, D. A. Peled, and A. Pretschner, editors, *Dependable Software Systems Engineering*, volume 40 of *NATO Science for Peace and Security Series, D: Information and Communication Security*, pages 1–25. IOS Press, 2015.

[2] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley. Enhancing symbolic execution with veritesting. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1083–1094, 2014.

[3] T. Ball and J. Daniel. Deconstructing dynamic symbolic execution. In *Proceedings of the 2014 Marktoberdorf Summer School on Dependable Software Systems Engineering*, 2015.

[4] R. Chugh, B. Hempel, M. Spradlin, and J. Albers. Programmatic and direct manipulation, together at last. *SIGPLAN Not.*, 51(6):341–354, June 2016.

[5] L. A. Clarke. A program testing system. In *Proc. of the 1976 annual conference*, pages 488–491, 1976.

[6] A. Desai, S. Gulwani, V. Hingorani, N. Jain, A. Karkare, M. Marron, S. R, and S. Roy. Program synthesis using natural language. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 345–356, New York, NY, USA, 2016. ACM.

[7] Google. Clusterfuzz - readme. `https://github.com/google/clusterfuzz/blob/2ae06a430c6f9bfcf418490f3416f28a94d51515/README.md`. Retrieved: June 2020.

[8] S. Gulwani. Programming by examples: Applications, algorithms, and ambiguity resolution. In *Proceedings of the 8th International Joint Conference on Automated Reasoning - Volume 9706*, pages 9–14, New York, NY, USA, 2016. Springer-Verlag New York, Inc.

[9] S. Gulwani, W. R. Harris, and R. Singh. Spreadsheet data manipulation using examples. *Commun. ACM*, 55(8):97–105, Aug. 2012.

[10] S. Gulwani and N. Jojic. Program verification as probabilistic inference. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, pages 277–289, New York, NY, USA, 2007. ACM.

[11] S. Gulwani and M. Marron. Nlyze: Interactive programming by natural language for spreadsheet data analysis and manipulation. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 803–814, New York, NY, USA, 2014. ACM.

[12] S. Gulwani, O. Polozov, and R. Singh. Program synthesis. *Foundations and Trends in Programming Languages*, 4(1-2):1–119, 2017.

[13] B. Hempel and R. Chugh. Semi-automated svg programming via direct manipulation. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, UIST '16, pages 379–390, New York, NY, USA, 2016. ACM.

[14] D. C. Ince. The automatic generation of test data. *The Computer Journal*, 30(1):63–69, 1987.

[15] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 215–224, New York, NY, USA, 2010. ACM.

[16] R. Joshi, G. Nelson, and K. Randall. Denali: A goal-directed superoptimizer. *SIGPLAN Not.*, 37(5):304–314, May 2002.

[17] G. Katz and D. Peled. Genetic programming and model checking: Synthesizing new mutual exclusion algorithms. In *Proceedings of the 6th International Symposium on Automated Technology for Verification and Analysis*, ATVA '08, pages 33–47, Berlin, Heidelberg, 2008. Springer-Verlag.

[18] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19:385–394, July 1976.

[19] V. Le, S. Gulwani, and Z. Su. Smartsynth: Synthesizing smartphone automation scripts from natural language. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '13, pages 193–206, New York, NY, USA, 2013. ACM.

[20] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. GenProg: A generic method for automated software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, 2012.

[21] LLVM Developer Group. libfuzzer. `http://llvm.org/docs/LibFuzzer.html`, 2016. Accessed March 20, 2019.

[22] Z. Manna and R. J. Waldinger. Toward automatic program synthesis. *Commun. ACM*, 14(3):151–165, Mar. 1971.

[23] S. Mechtaev, J. Yi, and A. Roychoudhury. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *International Conference on Software Engineering*, ICSE '16, pages 691–701, 2016.

[24] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, December 1990.

[25] MITRE. 2019 cwe top 25 most dangerous software errors. `https://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html`. Retrieved: June 2020.

[26] G. J. Myers. *Art of Software Testing*. Wiley, 1979.

[27] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 772–781, Piscataway, NJ, USA, 2013. IEEE Press.

[28] R. Padhye. *Abstractions and Algorithms for Specializing Dynamic Program Analysis and Random Fuzz Testing*. PhD thesis, EECS Department, University of California, Berkeley, Jul 2020.

[29] R. Padhye, C. Lemieux, K. Sen, L. Simon, and H. Vijayakumar. Fuzzfactory: domain-specific fuzzing with waypoints. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–29, 2019.

[30] O. Polozov and S. Gulwani. Flashmeta: A framework for inductive program synthesis. *SIGPLAN Not.*, 50(10):107–126, Oct. 2015.

[31] K. Sen, G. Necula, L. Gong, and W. Choi. Multise: Multi-path symbolic execution using value summaries. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 842–853, 2015.

[32] R. Singh and S. Gulwani. Transforming spreadsheet data types using examples. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 343–356, New York, NY, USA, 2016. ACM.

[33] A. Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, Berkeley, CA, USA, 2008. AAI3353225.

[34] N. Yaghmazadeh, C. Klinger, I. Dillig, and S. Chaudhuri. Synthesizing transformations on hierarchically structured data. *SIGPLAN Not.*, 51(6):508–521, June 2016.

[35] M. Zalewski. American fuzzy lop. `http://lcamtuf.coredump.cx/afl`, 2014. Accessed March 20, 2019.