Program Analysis

Jonathan Aldrich and Claire Le Goues

edited by Jonathan Aldrich for the China Summer International Program

Summer 2019

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 accelleration in Toyota cars, personal information stolen from Facebook by Cambridge Analytica, 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 analyiss 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 study 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
P	boolean predicates (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 predicates P as conditions. Arithmetic expressions a include variables x, numbers n, and one of several arithmetic operators (op_a). Predicates are represented by Boolean expressions that 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.

S	::=	$x := a$	P	::=	true	a	::=	x	op_b	::=	and | or
	|	skip		|	false		|	n	op_r	::=	$< \mid \leqslant \mid =$
	|	$S_1;\ S_2$		|	not P		|	$a_1\ op_a\ a_2$		|	$> \mid \geqslant$
	|	if P then S_1 else S_2		|	$P_1\ op_b\ P_2$				op_a	::=	$+ \mid - \mid * \mid /$
	|	while P do S		|	$a_1\ op_r\ a_2$						

[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` P `then` S_1 `else` S_2 would be translated into:

$1:$ if P then goto 4
$2:$ S_2
$3:$ goto 5
$4:$ S_1

Exercise 1. How would you translate a WHILE statement of the form `while` P `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]

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

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 maps conceptually to Nielsens' use of *labels* in the WHILE language specification in the textbook. This concept is akin to mapping line numbers to code.

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

Program analysis tools typically work on a representation of code 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 *judgement* as follows:

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

This means that given program state E, the expression e evaluates to n. This formulation is called *big-step* operational semantics; the $\Downarrow$ judgement 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:

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

Addition expressions illustrate a rule with premises:

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

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 judgement $\Downarrow$ as applied to statements and states therefore looks like:

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

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:

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

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

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

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:

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

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."

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

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 judgement rule like this for every language construct. These notes only include a subset for WHILE, for brevity.

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

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 ($\langle S, E\rangle$) 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 ($\langle \texttt{skip}, E\rangle$, in this case) if ever.[2] We write this new judgement using a slightly different arrow: $\rightarrow$. $\langle S, E\rangle \rightarrow \langle S', E'\rangle$ indicates one step of execution; $\langle S, E\rangle \rightarrow^* \langle S', E'\rangle$ indicates zero or more steps of execution. We formally define multiple execution steps as follows:

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

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 judgements are thus:

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

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:

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

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

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

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

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:

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

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

[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 includes 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:

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

We say that $\langle a, E\rangle \Downarrow n$ is *provable* (expressed mathematically as $\vdash \langle a, E\rangle \Downarrow n$) if there exists a well-formed derivation with $\langle a, E\rangle \Downarrow n$ 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 judgement 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:

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

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:

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

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 judgements 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 $D :: \langle S, E\rangle \Downarrow E'$ and induct on the structure of that derivation (we define $D ::$ Judgement to mean "D is the derivation that proves judgement." e.g., $D :: \langle x + 1, E\rangle \Downarrow 2$). 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 $\overline{S}$).
Inductive cases: For each derivation rule of the form

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

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 $D :: \langle x := 55, E_i\rangle \Downarrow E$, 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 $D :: \langle \text{while } P \text{ do } S, E_i\rangle \Downarrow E$, 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 $D :: \langle S, E\rangle \Downarrow E'$

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

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

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

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:

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

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

By inversion, and determinism of boolean expressions, D'' must also use the same `while-true` rule. So D'' must also have subderivations $D_2'' :: \langle S, E\rangle \Downarrow E_1''$ and $D_3'' :: \langle \texttt{while } P \texttt{ do } S, E_1''\rangle \Downarrow E''$. 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. 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:

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

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 $\sigma[x \mapsto Z]$ denotes dataflow information that is identical to σ except that the value in the mapping for x refers 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):

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

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:

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

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

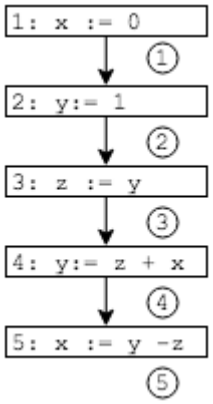
4.2 Running a dataflow analysis

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

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
1	Z		
2	Z	N	
3	Z	N	N
4	Z	N	N
5	$\top$	N	N

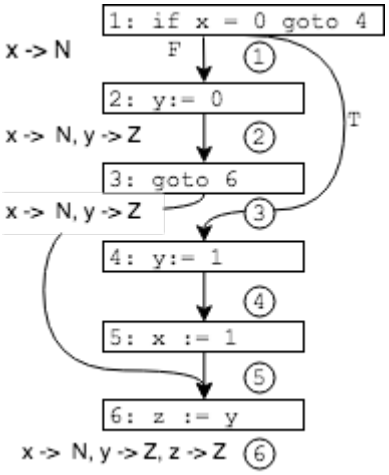
We simulate running the program in the analysis, using the flow function to compute, for each instruction in turn, the dataflow analysis information *after* the instruction from the information we had *before* the instruction. For such simple code, it is easy to track the analysis information using a table with a column for each program variable and a row for each program point (right, above). The information in a cell tells us the abstract value of the column's variable immediately after the instruction at that line (corresponding the the program points labeled with circles in the CFG).

Notice 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: Example

Things get more interesting in WHILE3ADDR code that contains `if` statements. In this case, there are two possible paths through the program. Consider the following simple example (left), and its CFG (middle). I have begun by analyzing one path through the program (the path in which the branch is not taken):

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

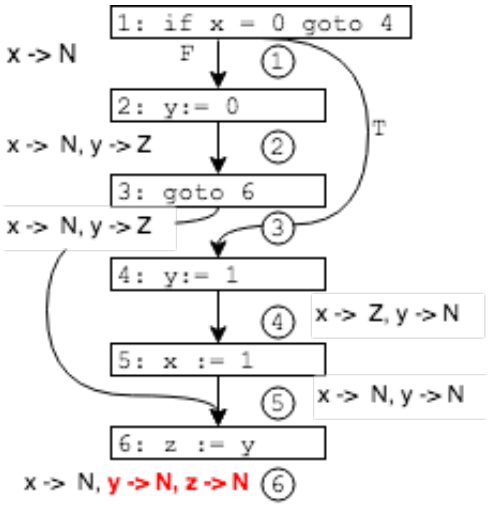


	x	y	z
1	Z_T, N_F		
2	N	Z	
3	N	Z	
4			
5			
6	N	Z	Z

In the table above, the entry for x on line 1 indicates the different abstract values produced for the true and false conditions of the branch. We use the false condition (x is nonzero) in analyzing instruction 2. Execution proceeds through instruction 3, at which point we jump to instruction 6. We have not yet analyzed a path through lines 4 and 5.

Turning to that alternative path, we can start by analyzing instructions 4 and 5 as if we had taken the true branch at instruction 1:

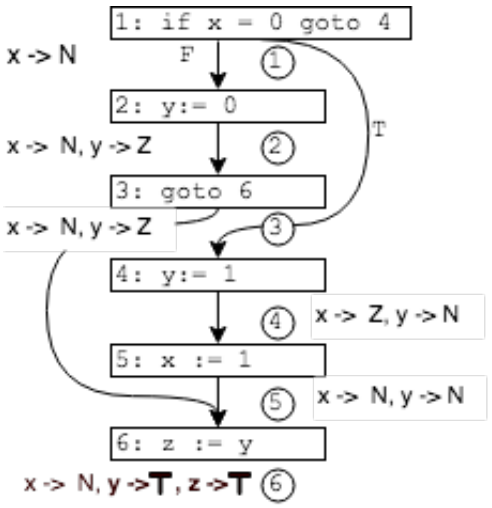
	x	y	z	
1	Z_T, N_F			
2	N	Z		
3	N	Z		
4	Z	N		
5	N	N		
6	N	Z	Z	*note: incorrect!*



We have a dilemma in analyzing instruction 6. We already analyzed it with respect to the previous path, assuming the dataflow analysis we computed from instruction 3, where x was nonzero and y was zero. However, we now have conflicting information from instruction 5: in this case, x is still nonzero, but y is also nonzero in this case.

We resolve this dilemma by combining the abstract values computed along the two paths for y and z. The incoming abstract values at line 6 for y are N and Z. We can represent this uncretainty with the abstract value $\top$, indicating that we do know know if y is zero or not at this instruction, because of the uncertainty about how we reached this program location. We can apply similar logic in the case of x, but because x is nonzero on both incoming paths we can maintain our knowledge that x is nonzero. Thus, we should reanalyze instruction 5 assuming the dataflow analysis information $\{x \mapsto N, y \mapsto \top\}$. The results of our final analysis are shown below:

	x	y	z	
1	Z_T, N_F			
2	N	Z		
3	N	Z		
4	Z	N		
5	N	N		
6	N	$\top$	$\top$	*corrected*



4.2.3 Join

We generalize the procedure of combining analysis results along multiple paths by using 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*. Any join-semilattice has a maximal element $\top$ (pronounced "top"). 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 have now introduced and considered all the elements necessary to define a dataflow analysis:

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

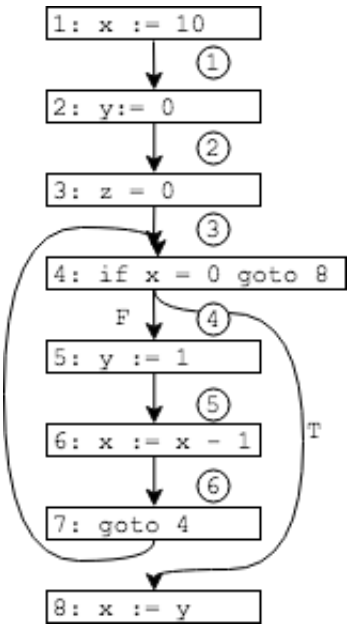
Note that the theory of lattices answers a side question that comes up when we begin analyzing the first program instruction: what should we assume about the value of input variables (like x on program entry)? If we do not know anything about the value x can be, a good choice is to assume it can be anything; That is, in the initial environment σ_0, input variables like x are mapped to $\top$.

4.2.4 Dataflow analysis of loops

We now consider WHILE3ADDR programs with loops. While an `if` statement produces two paths that diverge and later join, a loop produces an potentially unbounded number of pro-

gram paths. Despite this, we would like to analyze looping programs in bounded time. Let us examine how through the following simple looping example:[1]

1 : $x := 10$
2 : $y := 0$
3 : $z := 0$
4 : if $x = 0$ goto 8
5 : $y := 1$
6 : $x := x - 1$
7 : goto 4
8 : $x := y$



	x	y	z
1	N		
2	N	Z	
3	N	Z	Z
4	Z_T, N_F	Z	Z
5	N	N	Z
6	$\top$	N	Z
7	$\top$	N	Z
8			

The right-hand side above shows the straightforward straight-line analysis of the path that runs the loop once. We must now re-analyze instruction 4. This should not be surprising; it is analogous to the one we encountered earlier, merging paths after an `if` instruction. To determine the analysis information at instruction 4, we join the dataflow analysis information flowing in from instruction 3 with the dataflow analysis information flowing in from instruction 7. For x we have $N \sqcup \top = \top$. For y we have $Z \sqcup N = \top$. For z we have $Z \sqcup Z = Z$. The information for instruction 4 is therefore unchanged, except that for y we now have $\top$.

We can now choose between two paths once again: staying within the loop, or exiting out to instruction 8. We will choose (arbitrarily, for now) to stay within the loop, and consider instruction 5. This is our second visit to instruction 5, and we have new information to consider: since we have gone through the loop, the assignment $y := 1$ has been executed, and we have to assume that y may be nonzero coming into instruction 5. This is accounted for by the latest update to instruction 4's analysis information, in which y is mapped to $\top$. Thus the information for instruction 4 describes both possible paths. We must update the analysis information for instruction 5 so it does so as well. In this case, however, since the instruction assigns 1 to y, we still know that y is nonzero after it executes. In fact, analyzing the instruction again with the updated input data does not change the analysis results for this instruction.

A quick check shows that going through the remaining instructions in the loop, and even coming back to instruction 4, the analysis information will not 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. If we analyze instruction 6, for example, the input analysis information from instruction 5 is the same input analysis information we used when analyzing instruction 6 the last time around. Thus, instruction 6's output information will not change, and so instruction 7's input information will not change, and so on. No matter which instruction we run the analysis on, anywhere in the loop (and in fact before the loop), the analysis information will not change.

We say that the dataflow analysis has reached a *fixed point*.[2] In mathematics, a fixed point

[1] I provide the CFG for reference but omit the annotations in the interest of a cleaner diagram.
[2] Sometimes abbreviated in one word as fixpoint.

of a function is a data value v that is mapped to itself by the function, i.e. $f(v) = v$. In this 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 of the function for this loop, it is clear that further analysis of the loop will not be useful. Therefore, we will proceed to analyze statement 8. The final analysis results are as follows:

	x	y	z	
1	N			
2	N	Z		
3	N	Z	Z	
4	Z_T, N_F	$\top$	Z	*updated*
5	N	N	Z	*already at fixed point*
6	$\top$	N	Z	*already at fixed point*
7	$\top$	N	Z	*already at fixed point*
8	Z	$\top$	Z	

Quickly simulating a run of the program program shows that these results correctly approximate actual execution. The uncertainty in the value of x at instructions 6 and 7 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 imprecision in the analysis: this loop always executes at least once, so y will be nonzero. 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 in static analysis.

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.

4.2.5 A convenience: the $\perp$ abstract value and complete lattices

As we think about defining an algorithm for dataflow analysis more precisely, a natural question comes up concerning how instruction 4 is analyzed in the example above. On the first pass, we analyzed it using the dataflow information from instruction 3, but on the second pass we had to consider dataflow information from both instruction 3 and instruction 7.

It is more consistent to say that analyzing an instruction always uses the incoming dataflow analysis information from all instructions that could precede it. That way, we do not

have to worry about following a specific path during analysis. However, for instruction 4, this requires a dataflow value from instruction 7, even if instruction 7 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 least upper bounds $\sqcup$ and greatest lower bounds $\sqcap$ exist in L and are unique, is called a *complete lattice*.

The theory of $\bot$ and complete lattices provides an elegant solution to the problem mentioned above. We can initialize σ at every instruction 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.

4.3 Analysis execution strategy

The informal execution strategy outlined above considers all paths through the program, continuing until the dataflow analysis information reaches a fixed point. This strategy 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. This is sensible: 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 the analysis reaches a fixed point, the final result will be correct. The simplest correct algorithm for executing dataflow analysis can therefore be stated as follows:

```
for Instruction i in program
    input[i] = ⊥
input[firstInstruction] = initialDataflowInformation

while not at fixed point
    pick an instruction i in program
    output = flow(i, input[i])
    for Instruction j in sucessors(i)
        input[j] = input[j] ⊔ output
```

Although in the previous presentation we have been tracking the analysis information immediately after each instruction, it is more convenient when writing down the algorithm to track the analysis information immediately before each instruction. This avoids the need for a distinguished location before the program starts (the start instruction is not analyzed).

In the code above, the termination condition is expressed abstractly. It can easily be checked, however, by running the flow function on each instruction in the program. If the results of analysis do not change as a result of analyzing any instruction, then it has reached a fixed point.

How do we know the algorithm will terminate? The intuition is as follows. We rely on the choice of an instruction to be fair, so that each instruction is eventually considered. As long as the analysis is not at a fixed point, some instruction can be analyzed to produce new analysis 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 instruction, 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 as:

```
for Instruction i in program
    input[i] = ⊥
input[firstInstruction] = initialDataflowInformation
worklist = { firstInstruction }

while worklist is not empty
    take an instruction i off the worklist
    output = flow(i, input[i])
    for Instruction j in succs(i)
        if output ⋢ input[j]
            input[j] = input[j] ⊔ output
            add j to worklist
```

The algorithm above is very close to the generic algorithm declared previously, except for the worklist that chooses the next instruction to analyze and determines when a fixed point is reached.

We can reason about the performance of this algorithm as follows. We only add an instruction to the worklist when the input data to some node changes, and 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 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, $\sqcup$, or $\sqsubseteq$ test) 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.

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 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

5.1 Constant Propagation

While zero analysis was useful for simply tracking whether a given variable is zero or not, 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:

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

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

5.2 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:

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

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

Exercise 1. 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:

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

We would compute dataflow analysis information for the program shown above as follows:

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

5.3 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:

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

In this example, 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:

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

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

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. 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	$\{z\}$
7	3	$\{z\}$
3	6,2	$\{z, y\}$
6	5,2	$\{z, y\}$
5	4,2	$\{z, y\}$
4	3,2	$\{z, y\}$
3	2	$\{z, y\}$
2	1	$\{y\}$
1	$\varnothing$	$\{x\}$

Chapter 6

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:

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

In hte 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.

Note that this is not a truly precise syntactic specification (specifying "possibly empty list of arithmetic expressions" properly takes several intermediate syntactic steps), but providing one 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.

6.1 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:

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

Now what?

6.2 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.

6.3 Local vs. global variables

The above analyses assume we have only local variables. If we have 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.

6.4 Interprocedural Control Flow Graph

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:

```
1 :  fun double(x : int) : int
2 :      y := 2 * x
3 :      return y
4 :  fun main()
5 :      z := 5
6 :      w := double(z)
7 :      z := 10/w
8 :      z := 0
9 :      w := double(z)
```

What's the issue here?

6.5 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 all callees. But this works only 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 in a straightforward way, and in fact if we cache the results of analyzing fact(2) we can reuse this 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 can make the initial assumption that the result of the recursive call is $\bot$, which is 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 $analyzing : Stack[Context]$ ▷ 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 AnalyzeProgram                          ▷ starting point for interprocedural analysis
    worklist ← {Context(main, ⊤)}
    results[Context(main, ⊤)].input ← ⊤
    while NotEmpty(worklist) do
        ctx ← Remove(worklist)
        Analyze(ctx, results[ctx].input)
    end while
end function

function Analyze(ctx, σ_i)
    σ_o ← results[ctx].output
    Push(analyzing, ctx)
    σ'_o ←Intraprocedural(ctx, σ_i)
    Pop(analyzing)
    if σ'_o ⋢ σ_o then
        results[ctx] ← Summary(σ_i, σ_o ⊔ σ'_o)
        for c ∈ callers[ctx] do
            Add(worklist, c)
        end for
    end if
    return σ'_o
end function

function Flow(⟦n: x := f(y)⟧, ctx, σ_i)          ▷ called by intraprocedural analysis
    σ_in ← [formal(f) ↦ σ_i(y)]          ▷ map f's formal parameter to info on actual from σ_i
    calleeCtx ← GetCtx(f, ctx, n, σ_in)
    σ_o ←ResultsFor(calleeCtx, σ_in)
    Add(callers[calleeCtx], ctx)
    return σ_i[x ↦ σ_o[result]]              ▷ update dataflow with the function's result
end function

function ResultsFor(ctx, σ_i)
    σ ← results[ctx].output
    if σ ≠ ⊥ ∧ σ_i ⊑ results[ctx].input then
        return σ                                          ▷ existing results are good
    end if
    results[ctx].input ← results[ctx].input ⊔ σ_i ▷ keep track of possibly more general input
    if ctx ∈ analyzing then
        return ⊥                          ▷ initially optimistic assumption for recursive calls
    else
        return Analyze(ctx, results[ctx].input)
    end if
end function

function GetCtx(f, callingCtx, n, σ_i)
    return Context(f, σ_i)                  ▷ constructs a new Context with f and σ_i
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(); }
```

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

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

6.7 Termination

Under what conditions will context-sensitive interprocedural analysis terminate?

Consider the algorithm above. 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 $\bot$ 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.

6.8 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($f, callingCtx, n, \sigma_i$)
  return $Context(f)$
end function

Note that in this approach the same calling context might be used for several different input dataflow information σ_i, one for each call to GETCTX. This is handled correctly by RESULTS-FOR, 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 cannot be sensitive to additional contexts once the bound is reached, 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. 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). 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 so many times that performance becomes unacceptable. Thus multiple contexts must be combined somehow to reduce the number of times each function is 7 analyzed. The call-string approach provides an easy, but naive, way to do this: call strings can 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($f, callingCtx, n, \sigma_i$)
  $newStr \leftarrow$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 7

Pointer Analysis

7.1 Motivation for Pointer Analysis

In the spirit of extending our understanding of analysis to more realistic languages, consider programs with pointers, or variables whose value refers to another value stored elsewhere in memory by storing the address of that stored value. Pointers are very common in imperative and object-oriented programs, and ignoring them can dramatically impact the precision of other analyses that we have discussed. 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:

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

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:

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

7.2 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, as described above, computes the set *points-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 (PhD thesis: "Program Analysis and Specialization for the C Programming Language").

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 into four types: taking the address of a variable, copying a pointer from one variable to another, assigning through a pointer, and dereferencing a pointer:

$$\begin{array}{rcl} I & ::= & \ldots \\ & | & p := \&x \\ & | & p := q \\ & | & *p := q \\ & | & p := *q \end{array}$$

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 are used to improve the performance of the analysis, as precise pointer analysis can be notoriously expensive in practice.

We will formulate Andersen's analysis by generating set constraints which can later be processed by a set constraint solver using a number of technologies. Constraint generation for each statement works as given in the following set of rules. 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.

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

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

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

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

The constraints generated are all set constraints. 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 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.

A number of specialized set constraint solvers exist and constraints in the form above can be translated into the input for these. The dereference operation (the $*$ in $*p \supseteq q$) is not standard in set constraints, but it can be encoded—see Fähndrich's Ph.D. thesis for an example of how to encode Andersen's points-to analysis for the BANE constraint solving engine. 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. Note that in this example 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.

We can also apply Andersen's analysis to programs with dynamic memory allocation, such as:

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

In this example, 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. We can use the rules:

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

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.

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 from a theorem by David McAllester published in SAS'99. 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 $p \supseteq *q$ 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, Sradharan and Fink (SAS '09) showed that Andersen's algorithm can be executed in $O(n^2)$ time for *k-sparse* programs. The *k-sparse* assumption requires that at most k statements dereference each variable, and that the flow graph is sparse. They also show that typical Java programs are *k-sparse* and that Andersen's algorithm scales quadratically in practice.

7.2.1 Field-Insensitive Analysis

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:

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

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 $*$.

7.2.2 Field-Sensitive Analysis

In order to be more precise, we can 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.[1] We will no longer have inclusion constraints involving pointer dereferences, so we drop the *assign* and *dereference* rules. Instead we will generate inclusion constraints involving fields, using the two following rules:

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

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

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.

[1] note that in Java, the **`new`** expression plays the role of `malloc`

7.3 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 scalability in practice.

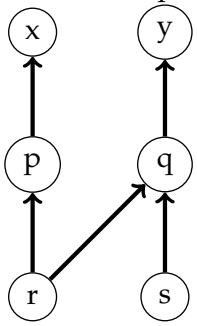
The first challenge in designing a near-linear time points-to analysis is to represent the results in linear space. This is nontrivial because over the course of program execution, any given pointer p could potentially point to the location of any other variable or pointer q. Representing all of these 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.

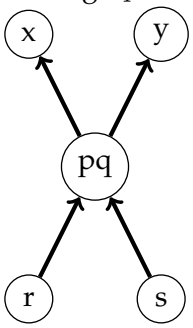
For example, consider the program below:

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

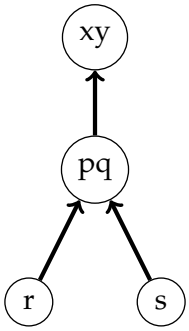
Andersen's points-to analysis would produce the following graph:



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:



To define Steensgaard's analysis more precisely, we will study a simplified version of that ignores function pointers. It can be specified as follows:

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

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

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

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

With each abstract location p, we associate the abstract location that p points to, denoted $*p$. Abstract locations are implemented as a union-find[2] 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:

[2]See any algorithms textbook

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

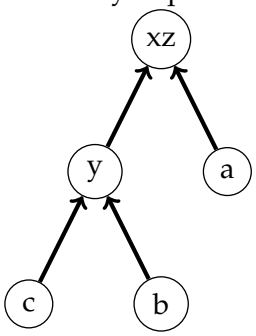
Once again, we implicitly invoke *find* on an abstract location before comparing it for equality, looking up the abstract location it points to, or calling *join* recursively.

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:

$$
\begin{array}{ll}
1: & a := \&x \\
2: & b := \&y \\
3: & \text{if } p \text{ then} \\
4: & \quad y := \&z \\
5: & \text{else} \\
6: & \quad y := \&x \\
7: & c := \&y
\end{array}
$$

His analysis produces the following graph for this program:



Rayside illustrates a situation in which Andersen must do more work than Steensgaard:

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

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. In contrast, 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 node. 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.[3]

[3]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.

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.

Chapter 8

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*

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

8.1.1 Assertion judgements using operational semantics

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

$$\begin{array}{rcllllll} A & ::= & \text{true} & \mid \text{false} & \mid e_1 = e_2 & \mid e_1 \geqslant e_2 & \mid A_1 \wedge A_2 \\ & \mid & A_1 \vee A_2 & \mid A_1 \Rightarrow A_2 & \mid \forall x.A & \mid \exists x.A & \end{array}$$

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 $E \models A$, read "A is true in E". The $\models$ judgment is defined inductively on the structure of assertions, and relies on the operational semantics of WHILE arithmetic expressions. For example:

$$\begin{array}{ll} E \models \text{true} & \textit{always} \\ E \models e_1 = e_2 & \textit{iff } \langle e_1, E\rangle \Downarrow n = \langle e_2, E\rangle \Downarrow n \\ E \models e_1 \geqslant e_2 & \textit{iff } \langle e_1, E\rangle \Downarrow n \geqslant \langle e_2, E\rangle \Downarrow n \\ E \models A_1 \wedge A_2 & \textit{iff } E \models A_1 \text{ and } E \models A_2 \\ \ldots & \\ E \models \forall x.A & \textit{iff } \forall n \in \mathcal{Z}.E[x := n] \models A \\ E \models \exists x.A & \textit{iff } \exists n \in \mathcal{Z}.E[x := n] \models A \end{array}$$

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

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

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 $\forall x.A$ assertion (check every integer?!). This motivates a new symbolic technique for deriving valid assertions from others that are known to be valid.

8.1.2 Derivation rules for Hoare triples

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

$$\frac{\vdash A \quad \vdash B}{\vdash A \wedge B}\ and$$

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?):

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

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

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:

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

We elide a formal statement of the soundness of this system. Intuitively, it expresses that the axiomatic proof we can derive using these rules is equivalent to the operational semantics derivation (or that they are sound and relatively complete, that is, as complete as the underlying logic).

8.2 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.

8.2.1 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]

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

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:

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

8.2.2 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.
- $(Inv \wedge \neg B) \Rightarrow Q$: 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.

[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.

8.2.3 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:

$$\{P\}\ S_{inv}\ \{Q\}$$

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 $wp(S_{annot}, Q)$ but still weaker than P: $P \Rightarrow VC(S_{annot}, Q) \Rightarrow wp(S_{annot}, Q)$ We compute VC using a verification condition generation procedure $VCGen$, which mostly follows the definition of the wp function discussed above:

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

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

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

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:

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

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 9

Symbolic Execution

9.1 Symbolic Execution Overview

Symbolic execution is a way of executing a program abstractly, so that one abstract execution covers multiple possible inputs of the program that share a particular execution path through the code. The execution treats these inputs symbolically, "returning" a result that is expressed in terms of symbolic constants that represent those input values.

Symbolic execution is less general than abstract interpretation, because it doesn't explore all paths through the program. However, symbolic execution can often avoid approximating in places where AI must approximate in order to ensure analysis 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.

9.1.1 A Generalization of Testing

As the above discussion suggests, symbolic execution is a way to generalize testing. A test involves executing a program concretely on one specific input, and checking the results. In contrast, symbolic execution considers how the program executes abstractly on a family of related inputs. 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);
```

Running this code with a = 1, b = 2, and c = 1 causes the assertion to fail, and if we are good (or lucky) testers, we can stumble upon this combination and generalize to the combination of input spaces that will lead to it (and hopefully fix it!).

Instead of executing the code on concrete inputs (like a = 1, b = 2, and c = 1), symbolic execution evaluates it on *symbolic inputs*, like $a = \alpha, b = \beta, c = \gamma$, and then tracks execution in terms of those symbolic values. 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. After a given symbolic execution

is complete, the engine may go back to the branches taken and explore other paths through the program.

To get an intuition for how symbolic analysis works, consider abstractly executing a path through the program above. As we go along the path, we will keep track of the (potentially symbolic) values of variables, and we will also track 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. 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!

9.1.2 History of Symbolic Analysis

Symbolic execution was originally proposed in the 1970s, 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 4 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.

9.2 Symbolic Execution Semantics

We can write rules for evaluating programs symbolically in WHILE. We will write the rules in a style similar to the big-step semantics we wrote before, but incorporate symbolic values and keep track of the path conditions we have taken.

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:

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

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

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

Now we can define big-step rules for the symbolic evaluation of expressions, resulting 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:

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

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

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

We can likewise define rules for statement evaluation. These rules need to update not only the environment E, but also a path guard g:

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

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

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

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

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

The rules for skip, sequence, and assignment are compositional in the expected way, with the arithmetic expression on the right-hand side of an assignment evaluating to a symbolic expression rather than a value. The interesting rules are the ones for if. 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.

We leave the rule for `while` to the reader, following the principles behind the `if` rules above.

9.3 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:

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

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:

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

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

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

9.4 Symbolic Execution Implementation and Industrial Use

Of course programs with loops have infinite numbers of paths, so exhaustive symbolic execution is not possible. Instead, tools take hueristics, such as exploring all execution trees down to a certain depth, or limiting loop iteration to a small constant, or trying to find at least one path that covers each line of code in the program. In order to avoid analyzing complex library code, symbolic executors may use an abstract model of libraries.

Symbolic execution has been used in industry for the last couple of decades. One of the most prominent examples is the use of the PREfix to find errors in C/C++ code within Microsoft [3].

Chapter 10

Program Synthesis

Note: A complete, if lengthy, resource on inductive program synthesis is the book "Program Synthesis" by Gulwani *et. al* [10]. 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.

10.1 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 [6] (often with some kind of user- or synthesizer-driven interaction), traces, natural language [5, 9, 15], or full- or partial programs [22]. 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., [17]), which maps a high-level (e.g. logical) specification to an executable implementation. 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. We will (very) briefly overview Denali [13], a prototypical deductive synthesis technique, using slides. 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 [8], or genetic programming [14]. Alternatively, techniques based on logical reasoning delegate the search problem to a constraint solver. We will spend more time on this set of techniques.

10.2 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) [7], data extraction/processing/wrangling [6, 21], layout transformation of tables or tree-shaped structures [23], graphics (constructing structured, repetitive drawings) [11, 4], program repair [18, 16] (spoiler alert!), superoptimization [13], and efficient synchronization, among others.

Inductive synthesis consists of several family of approaches; we will overview several prominent examples, without claiming to be complete.

10.2.1 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) [22, 20]. 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.

10.2.2 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...

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.

10.3 Oracle-guided Component-based Program Synthesis

Problem statement and intuition.[1] 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 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	$z_1 := \texttt{input}^1$
$\ldots$	$\ldots$
m	$z_m := \texttt{input}^m$
$m+1$	$z_{m+1} := f_?(z_?, \ldots, z_?)$
$m+2$	$z_{m+2} := f_?(z_?, \ldots, z_?)$
$\ldots$	$\ldots$
$m+n$	$z_{m+n} := f_?(z_?, \ldots, z_?)$
$m+n+1$	$\texttt{return}\ z_?$

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:

[1] These notes are inspired by Section III.B of Nguyen *et al.*, ICSE 2013 [19] ...which provides a really beautifully clear exposition of the work that originally proposed this type of synthesis in Jha *et al.*, ICSE 2010 [12].

$$L \;:=\; \{l_x | x \in Q \cup R \cup \overrightarrow{Y} \cup r\}$$

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:

$$\psi_{wfp}(L,Q,R) \stackrel{def}{=} \bigwedge_{x \in Q}(0 \leqslant l_x < M) \;\wedge\; \bigwedge_{x \in R}(|\overrightarrow{Y}| \leqslant l_x < M) \;\wedge\; \psi_{cons}(L,R) \wedge \psi_{acyc}(L,Q,R)$$

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 11

Satisfiability Modulo Theories

11.1 Motivation: Tools to Check Hoare Logic Specifications

Recall the lectures on Hoare Logic. We use weakest preconditions to generate a formula 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 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'd like tools to check this automatically. That won't be feasible for all formulas, but as we will see it is feasible for a useful subset of formulas.

11.2 The Concept of Satisfiability Modulo Theories

First, let's reduce 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. There are a lot of satisfiability modulo theories (SMT) solvers that do this.

What does the "modulo theories" part of SMT mean? Well, strictly speaking satisfiability is for boolean formulas: formulas 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. $+, >$), 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. We'll see how this works later, but first let's look at how we can check ordinary satisfiability.

11.3 DPLL for Satisfiability

The DPLL algorithm, named for its developers Davis, Putnam, Logemann, and Loveland, is an efficient approach to solving boolean satisfiability problems. To use DPLL, we will take a formula F and transform it 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:

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

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

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

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 in order 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:

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

Now here we can see that b always occurs positively (i.e. without a $\neg$ in front of it within a CNF formula). If we choose 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 c always occurs negatively, i.e. in the form $\neg c$. We say that a literal that occurs only positively, or only negatively, in a formula is *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:

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

which is equivalent to false, so 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 φ = 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
        φ ← UNIT-PROPAGATE(l, φ)
    end for
    for all literals l occurring pure in φ do
        φ ← PURE-LITERAL-ASSIGN(l, φ)
    end for
    l ← CHOOSE-LITERAL(φ)
    return DPLL(φ ∧ l) ∨ DPLL(φ ∧ ¬l)
end function
```

Mostly the algorithm above is straightforward, but there are a couple of notes. First of all, the algorithm does unit propagation before pure literal assignment. Why? Well, it's good to do unit propagation first, because doing so 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. This is because pure literal assignment can eliminate entire clauses but it 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.

11.4 Solving SMT Problems

How can we solve a problem that involves various theories, in addition to booleans? Consider a conjunction of the following formulas:[1]

$$f(f(x) - f(y)) = a$$
$$f(0) = a + 2$$
$$x = y$$

This problem mixes linear arithmetic with the theory of uninterpreted functions (here, f is some unknown function). The first step in the solution is to separate the two 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 first formula, we'd like to factor out the subtraction, so we generate a fresh variable and divide the formula into two:

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

$f(e1) = a$ *// in the theory of uninterpreted functions now*
$e1 = f(x) - f(y)$ *// still a mixed formula*

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

$e1 = e2 - e3$ *// in the theory of arithmetic now*
$e2 = f(x)$ *// in the theory of uninterpreted functions*
$e3 = f(y)$ *// in the theory of uninterpreted functions*

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

$$f(e4) = e5$$
$$e4 = 0$$
$$e5 = a + 2$$

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

$$f(e1) = a$$
$$e2 = f(x)$$
$$e3 = f(y)$$
$$f(e4) = e5$$
$$x = y$$

And second, formulas in the theory of arithmetic:

$$e1 = e2 - e3$$
$$e4 = 0$$
$$e5 = a + 2$$
$$x = y$$

Notice that $x = y$ is in both sets of formulas. In SMT, we use the fact that equality is something that every theory understands...more on this in a moment. For now, 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:

$$e1 = e2 - e3$$
$$e4 = 0$$
$$e5 = a + 2$$
$$x = y$$
$$e2 = e3$$
$$a = e5$$

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:

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

(note: if we had multiple theories, I am assuming we've already added the equality constraints between them, as described above)

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 we can run a SAT solver using the DPLL algorithm. 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. This combination of DPLL with a theory T is called DPLL-T.

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 12

Concolic Testing

12.1 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 a set of test cases that covers all of the source code, as well as finding inputs that lead to difficult-to-trigger corner case defects.

Symbolic execution, discussed in the last lecture, 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.

Concolic testing overcomes these problems by combining **conc**rete execution (i.e. testing) with symb**olic** execution.[1] 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.

12.2 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] The word concolic is a portmanteau of **conc**rete and symb**olic**

[2] This formulation is due to Wolfram Schulte

```
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;
```

12.3 Overview

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  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.

12.4 Implementation

Ball and Daniel [2] 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.

12.5 Acknowledgments

The structure of these notes and the examples are adapted from a presentation by Koushik Sen.

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. Ball and J. Daniel. Deconstructing dynamic symbolic execution. In *Proceedings of the 2014 Marktoberdorf Summer School on Dependable Software Systems Engineering*, 2015.

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

[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] 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.

[6] 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.

[7] S. Gulwani, W. R. Harris, and R. Singh. Spreadsheet data manipulation using examples. *Commun. ACM*, 55(8):97–105, Aug. 2012.

[8] 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.

[9] 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.

[10] S. Gulwani, O. Polozov, and R. Singh. Program synthesis. *Foundations and Trends in Programming Languages*, 4(1-2):1–119, 2017.

[11] 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.

[12] 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.

[13] R. Joshi, G. Nelson, and K. Randall. Denali: A goal-directed superoptimizer. *SIGPLAN Not.*, 37(5):304–314, May 2002.

[14] 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.

[15] 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.

[16] 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.

[17] Z. Manna and R. J. Waldinger. Toward automatic program synthesis. *Commun. ACM*, 14(3):151–165, Mar. 1971.

[18] 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.

[19] 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.

[20] O. Polozov and S. Gulwani. Flashmeta: A framework for inductive program synthesis. *SIGPLAN Not.*, 50(10):107–126, Oct. 2015.

[21] 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.

[22] A. Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, Berkeley, CA, USA, 2008. AAI3353225.

[23] N. Yaghmazadeh, C. Klinger, I. Dillig, and S. Chaudhuri. Synthesizing transformations on hierarchically structured data. *SIGPLAN Not.*, 51(6):508–521, June 2016.