Notes on Program Analysis

Michael D. Ernst

April 5, 2024

Contents

Chapter 1

Introduction

Suppose you have written a program. What can you know about it? A *program analysis* tool provides the answers to questions such as:

- Does my program implement its specification? That is, is my program correct?
- What does a given program do?
- What method should an IDE suggest during autocompletion?
- What is the best name for a method?
- Which lines of code are most likely to be defective?
- When changing one line of code, what other lines might also need to be changed?

Program analysis is essential to enforcing security, achieving correctness, improving performance, and reducing costs when developing and maintaining software.

This book is about research techniques and ideas in program analysis. This book aims to partially answer the questions

- What can I know about a program?
- How can I know it?

Program analysis can be divided into static analysis and dynamic analysis. A *static analysis* examines the program's code, but it does not run the program. Examples include type checking and compilation. A *dynamic analysis* observes program executions. Examples include testing and profiling.

Most questions about a program can be answered using either static analysis or dynamic analysis — sometimes even using the same algorithms. Static analyses and dynamic analyses have different tradeoffs. One of the themes of this book is the relationship between static and dynamic analyses. For a discussion, see the short paper "Static and dynamic analysis: synergy and duality" [5]. Other themes of the book are the tradeoff between precision and performance; the tradeoff between power and transparency; and how to tune an analysis to the real problem that programmers suffer.

Some program analysis topics include:

- abstract interpretation, also known as dataflow analysis or symbolic execution, which is the classic static analysis; see chapter 2
- type systems, where tasks include type checking, type inference, and application of types to non-standard problems, such as aliasing [1, 9]
- verification, which gives a proof about all possible executions of a program and is needed in mission-critical situations

- model checking, which is exhaustive exploration of some space of possibilities; the possibilities may be kept track of explicitly or symbolically, and a key challenge is abstraction
- testing, where tasks include test generation (of both inputs and oracles), selection, and prioritization; see chapter 3
- other dynamic analyses such as profiling
- debugging
- refactoring
- analysis back ends and decision procedures: sometimes the best way to solve a program analysis task is to convert it into some other form and use an existing solver

Chapter 2

Abstract interpretation

Abstract interpretation is a way of estimating what a program might do at run time. Abstract interpretation is a *static analysis* because it examines the program's source code, but it does not run the program. (An analysis that runs the program, such as testing, profiling, or debugging, is a *dynamic analysis*.) Abstract interpretation can answer many questions about a program. For example, it can guarantee lack of null pointer exceptions by asking, for every dereferenced variable x, "Is variable x guaranteed to be non-null on every possible execution?"

Abstract interpretation can be explained by contrast to regular program execution.

- Running a program produces a concrete output; depending on the input, on one run it might be the number 22 and on another run it might be the number 42. Every expression in the program evaluates to a concrete value such as `1`, `-1`, or `"Hello world"`, and every variable holds such a value. See fig. 2.1 for an example. The figure shows the concrete store after each statement. A store is a mapping from variables to values.

- Abstract interpretation simulates program execution, but each expression evaluates to, and each variable holds, a *property*. Example properties are "an even number" and "a positive number". See fig. 2.1 for an example, where the three possible properties are "an even number", "an odd number", and "unknown". The figure shows the concrete store and the abstract store after each statement.

Abstract interpretation *approximates* a program's execution. It computes an estimate of properties that are true over all possible executions of the program, but without executing the program infinitely many times. An abstract interpretation is *sound* if every property that it reports is true of every possible execution. (Not every abstract interpretation is sound, but most are. We will assume an abstract interpretation is sound unless otherwise indicated.) However, abstract interpretation might be imprecise; for example, it is always sound to report "unknown".

The central challenge for the discipline of static analysis is choosing an appropriate abstraction: one that is simple enough for efficient computation, but expressive enough to retain precision.

In the name "abstract interpretation", "abstract" means that each value is a property (also known as an abstract value) rather than a concrete value, and "interpretation" means that the program is symbolically executed over those abstract values rather than concrete values. Other names for abstract interpretation are "symbolic execution" and "dataflow analysis".

2.1 The components of an abstract interpretation

An abstract interpretation consists of two parts:

1. The **lattice**. This consists of the set of abstract values, plus the relationships among them indicating which properties are stronger than others. For example, "divisible by 4" is a stronger property than "even", and "even" is stronger than "unknown". Earlier, we called the abstract values "properties".

```
                          x = undef, y = undef          x = undef, y = undef
x = 0;
                          x = 0, y = undef              x = even, y = undef
y = read_even();
                          x = 0, y = 8                  x = even, y = even
x = y + 1;
                          x = 9, y = 8                  x = odd, y = even
y = 2 * x;
                          x = 9, y = 18                 x = odd, y = even
x = y - 2;
                          x = 16, y = 18                x = even, y = even
y = x / 2;
                          x = 16, y = 8                 x = even, y = unknown

Program                   Concrete execution            Abstract execution
```

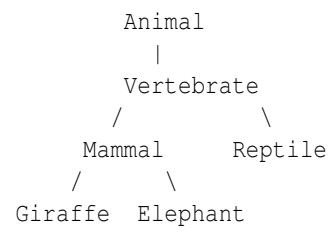
Figure 2.1: Contrasting concrete and abstract execution. The center column shows concrete stores, each one a mapping from variables to concrete values such as 8 or 9. The right column shows abstract stores, each one a mapping from variables to abstract values such as even, odd, or unknown.

2. The **transfer functions**. These tell how to perform computations on abstract values. The rules of concrete arithmetic say that $1 + 2 = 3$. The rules of abstract arithmetic (that is, the transfer functions) might say that $\mathsf{odd} + \mathsf{even} = \mathsf{odd}$. Transfer functions are sometimes called "flow functions".

2.2 Lattices

A lattice represents information about an expression. More specifically, each point in the lattice represents a different estimate about a run-time value (such as the value of a variable or the value to which an expression evaluates).

A lattice is analogous to a type hierarchy and is written the same way. For example:

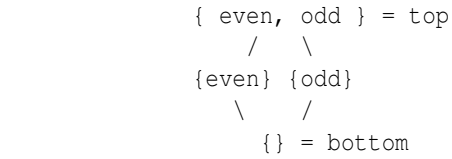


(This hierarchy is not actually a lattice because it lacks a bottom element, but the intuition holds.)

Here are three equivalent ways to view either a subtyping relationship or the lattice relationships, $\sqsubseteq$.

- Each lower point is-a instance of a higher point.
  For example, every mammal is-a vertebrate.
- Higher in the lattice, the set of possible values is larger.
  For example, the set of all vertebrates includes the set of all mammals.
- Higher in the lattice, the properties of the elements (or constraints on which elements are in the set) are weaker; lower in the lattice, the properties are stronger.
  For example, all mammals have 7 neck bones, but that is not true of all vertebrates.

Here is another example of a lattice. Each point in the lattice is a set.



Top ($\top$) is used when the estimate includes all possible values. Bottom ($\bot$) is used when the estimate includes no possible values. Bottom represents dead code and infinite loops such as the value of f(7) where def f(x) = f(x). Bottom may also represent uninitialized variables.

Sometimes, we are sloppy and write even rather than $\{\mathsf{even}\}$; if you see even used in a context that requires a set, then it means $\{\mathsf{even}\}$.

2.2.1 The components of a lattice

A lattice consists of two parts:

1. A *domain*: a set of points.

   It is also essential to know the meaning of each point: what run-time possibilities it represents. This is important for humans who are implementing or reasoning about the analysis. A mapping from the meaning of a point to the set of run-time possibilities is called a "concretezation function". It is not used when performing abstract interpretation, but is used in proofs about abstract interpretation.

   In the example immediately above, the domain consists of the 4 points $\{\}$, $\{\mathsf{even}\}$, $\{\mathsf{odd}\}$, and $\{\mathsf{even},\mathsf{odd}\}$. Each point is a set of properties, all of which are known to be true. Note that the domain is the powerset of $\{\mathsf{even},\mathsf{odd}\}$; such powerset domains are common.

   Alternatively, a point can be represented by a set of possible values, in which case the 4 points are $\{\}$, $\{\ldots,-2,0,2,\ldots\}$, $\{\ldots,-3,-1,1,3,\ldots\}$, and $\{\ldots,-2,-1,0,1,2,\ldots\}$. Since one definition of a type is a set of values, there is a strong connection between abstract interpretation and type theory.

2. An ordering or hierarchy among the elements. It can be expressed as

   - a less-than relation $\sqsubset$, or
   - a least-upper-bound (lub) operation $\sqcup$. The lub of two lattice points is the unique lowest point that is at least as great as both of them. For example, $\{\mathsf{even}\} \sqcup \{\mathsf{odd}\} = \top$, and $\{\mathsf{even}\} \sqcup \bot = \{\mathsf{even}\}$. An "upper bound" is a value that is at least as great. The least upper bound is the upper bound that is lowest.

   Given either one of these, the other one is uniquely determined.

   When drawing a lattice, the ordering is represented by lines, with larger points higher in the diagram.

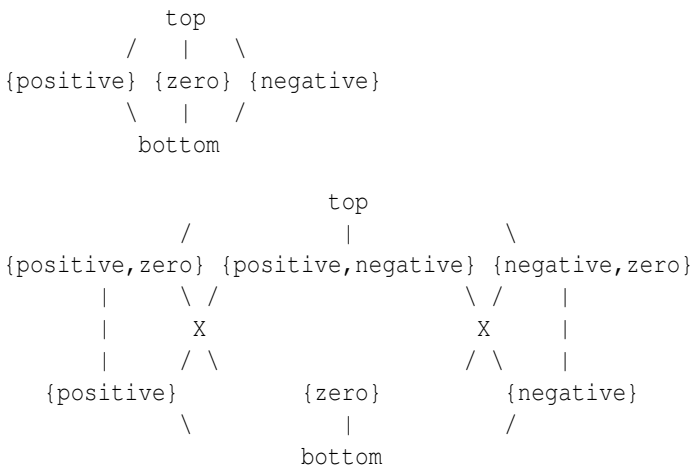
   The less-than relation need not be total or complete. For two points e_1 and e_2, it is possible that neither $e_1 \sqsubset e_2$ nor $e_2 \sqsubset e_1$ is true.

   If $x \sqsubset y$, then we say that x is lower or less, and that y is higher or greater.

   The lub function represents generalization. It is used when there are multiple possibilities, to create a single abstract value that encompasses the possibilities. In a program, lub is used at a *join point*, where two threads of control meet. An example of a join point is immediately after an if statement, which can be reached via the then branch or the else branch.

   If on the then branch variable x is a giraffe and on the else branch x is an elephant, then afterward x is a mammal because giraffe $\sqcup$ elephant = mammal. Similarly, giraffe $\sqcup$ reptile = vertebrate and giraffe $\sqcup$ mammal = mammal.

Here are two more example lattices:



The second example is the powerset lattice, similar to the even-odd example above.

Exercise 1 *What are the tradeoffs between these two lattices?*

A well-formed lattice must satisfy the following properties.

- Any two points in the lattice have a unique least upper bound.
- The least upper bound operator must be monotonic. A unary function f is monotonic if $\forall a,b.a \sqsubseteq b \Rightarrow f(a) \sqsubseteq f(b)$. A binary function f is monotonic if $\forall a,b,c,d.a \sqsubseteq b \wedge c \sqsubseteq d \Rightarrow f(a,c) \sqsubseteq f(b,d)$.
- The lattice has no infinite ascending chains. That is, if you start at any point and follow lines (*sqsubset* relationships) upward, you will get to $\top$ in a finite number of steps. (This property is not strictly necessary; see section 2.17.)

Relationship to type theory

Abstract interpretation has a strong relationship to type theory.

A type can be characterized as a set of values. Likewise for each property in the abstract domain; for example, the property even represents the set $\{\ldots,-2,0,2,4,\ldots\}$.

In either type theory or abstract interpretation, the main design challenge is choosing the abstractions or types. Types are designed to prevent run-time corruption and to prevent programmer mistakes. Each abstract domain is designed for some specific purpose, and its abstraction may be more or less precise than the programming language's built-in types.

To prove that a program has no out-of-bounds array accesses, an abstract domain could distinguish between negative and non-negative integers, because every negative integer is an illegal index. This is a finer-grained distinction than the programming language, which typically has a single type for all integers.

To prove that a program has no null pointer exceptions, an abstract domain could distinguish between null and non-null references. However, it would not distinguish between references to different types, such as a reference to a list vs. a reference to a set.

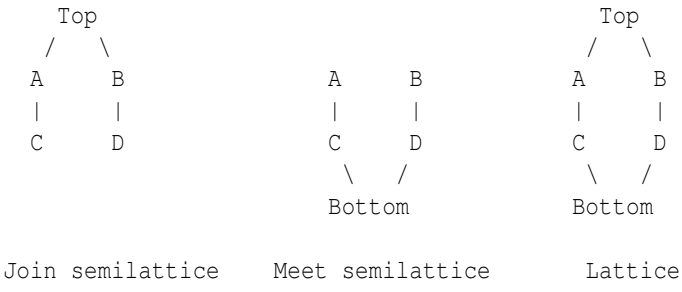
In a programming language, each variable typically has a single type throughout its scope. By contrast, in an abstract interpretation, the abstract variable of a type can change at an assignment or at a boolean test. Flow-sensitive type systems do exist [10], and these type systems have many similarities to dataflow analysis.

Abstract interpretation also has strong relationships to other analyses. Hoare Logic (a form of program verification) can be viewed as abstract interpretation with arbitrary symbolic expressions as the abstract values, and with the values written explicitly at each program point. Other types of verification, such as conversion to a set of constraints that are solved by an external solver such as a SAT solver, do not have such an obvious relationship. However, it is a theorem that any static analysis can be expressed as an abstract interpretation, and abstract interpretation is no stronger or weaker than any other static analysis.

Formal definition of a lattice

A lattice is a partially ordered set (that is, a set together with an ordering relation $\sqsubseteq$) in which every pair of elements has a least upper bound and a greatest lower bound.

Here is a sequence of definitions culminating in the definition of a lattice:



Join semilattice Meet semilattice Lattice

set unordered collection of distinct elements

partially ordered set Has a binary relationship $\sqsubseteq$ that is

> **reflexive** $x \sqsubseteq x$
>
> **anti-symmetric** $x \sqsubseteq y \wedge y \sqsubseteq x \rightarrow x = y$
>
> **transitive** $x \sqsubseteq y \wedge y \sqsubseteq z \rightarrow x \sqsubseteq z$
>
> The fact that the set is *partially* ordered means that it is possible that some elements have no relationship with one another, that is, $\exists x, y : x \not\sqsubseteq y \wedge y \not\sqsubseteq x$. Two examples are giraffe and elephant, and giraffe and reptile.

join semilattice partially ordered set in which each pair of elements has a unique least upper bound (also known as a "join")

meet semilattice partially ordered set in which each pair of elements has a unique greatest lower bound (also known as a "meet")

lattice both a join semilattice and a meet semilattice

For more details, see `https://en.wikipedia.org/wiki/Lattice_(order)`.

If the lub is not unique, then you would have to make an arbitrary choice when computing $x \sqcup y$, and the choices could violate properties such as associativity: $(x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$.

2.2.2 Transfer functions

In order to *run* a program that contains the expression `x + y`, the computer must have a definition of the + operator over the concrete numbers, because variables `x` and `y` hold concrete numbers at run time. In order to *perform abstract interpretation* on a program that contains the expression `x + y`, the abstract interpretation must have a definition of the + operator over the abstract values that are estimates for the run-time values held in variables `x` and `y`.

To make this more concrete, suppose that you know that x is even and y is even. What can you say about the value of $x + y$? How do you know? The abstract interpretation needs that same knowledge.

A transfer function represents an operation in the program, such as + or *. Here is an addition table for the properties "even" and "odd".

+	even	odd
even	even	odd
odd	odd	even

To make it a transfer function, each item should be a member of the domain:

+	{even}	{odd}
{even}	{even}	{odd}
{odd}	{odd}	{even}

The above transfer function is incomplete, however! The domain of the even/odd lattice contains 4 points, not just 2. The full table needs to account for the top and bottom values:

+	{even}	{odd}	{even,odd}	{}
{even}	{even}	{odd}	{even,odd}	{}
{odd}	{odd}	{even}	{even,odd}	{}
{even,odd}	{even,odd}	{even,odd}	{even,odd}	{}
{}	{}	{}	{}	{}

Thinking of each input as a set of values can help you see why these are the right values for the transfer function.

For brevity, we sometimes abuse notation by writing the transfer function as:

+	even	odd	$\top$	$\bot$
even	even	odd	$\top$	$\bot$
odd	odd	even	$\top$	$\bot$
$\top$	$\top$	$\top$	$\top$	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

An abstract interpretation needs a transfer function for every operation and statement in the programming language. This book generally uses a simple language with these constructs:

```
v=5
v=v2
x= v2 + v3
x= v2 * v3
...
if (v) then s else s
goto
```

In our simple language, the condition of an if statement is always a variable, and the right-hand side of an assignment is always an expression with only one operator. This is also called *3-address form*.

Three-address form simplifies a complex expression into multiple simple variable assignments, where the right-hand side uses at most one operation and two expressions. (There are at most 3 variables, thus the name "3-address form", where an "address" represents where the variable values are stored.) For example, consider the Pythagorean formula:

```
h = sqrt(a*a + b*b)
```

This would be represented in 3-address form as

```
tmp1 = a*a
tmp2 = b*b
tmp3 = tmp1 + tmp2
h = sqrt(tmp3)
```

Using 3-address form during analysis means that the analysis writer only has to define the analysis rules for simple expressions with a single operation, rather than worrying about how to abstractly execute more complicated expressions. For more details, see `https://en.wikipedia.org/wiki/Three-address_code`.

2.3 Summary: the parts of an abstract interpretation

An abstract interpretation consists of:

- lattice
- transfer functions (which would be better named, "abstract operations")

The lattice consists of:

- points, together with the meaning of each. (The concretization function maps from each point to the set of run-time values that it represents or is an estimate for, but it is not used during the process of abstract interpretation.)
- relationships between them, expressed via either a $\sqsubset$ relation or a lub function.

If we define the lattice in terms of lub, then the lub function:

- must be complete, with a unique value for every set of arguments
- must be monotonic

Exercise 2 *What are the corresponding requirements on the partial order $\sqsubset$? Can we express it without redefining lub? Is this the reason that formalisms usually define the lattice in terms of the lub rather than the partial order $\sqsubset$?*

2.4 More transfer functions

Consider this code:

```
x = 0;
y = read_even_integer();   // produces an even integer
x = y+1;
y = 2*x;
x = y-2;
y = x/2;
```

Before proceeding, try to perform abstract evaluation, using the even/odd lattice as defined so far, and with transfer functions for *, -, and / to complement that for +.

You should have gotten stuck on the first line! (If you didn't, that is OK, but for the future remember not to use your own smarts when simulating an algorithm.) The transfer functions show how to apply the + operation to two abstract values, but what abstract value corresponds to the integer value 0?

An "abstraction function" maps from concrete values to abstract values. For instance, it says that 0 is even, 1 is odd, and so forth. It is used most frequently for manifest constants (that is, literals) in source code. It can be viewed as a transfer function from constants to abstract values. (The dual is a "concretezation function" that maps from abstract value to the set of concrete values it represents. This notion is used in proofs, for example in establishing that a "Galois connection" exists.)

Here are some varieties of transfer functions:

- abstraction functions for constants (a transfer function of arity 0, since the value doesn't depend on any previous abstract value); that is, a mapping from concrete domain to abstract domain
- summaries for library routines, such as read_even_value() — see section 2.13
- programming language operations, such as + and *

2.5 Choosing a domain

The main challenge in creating a static analysis is choosing an appropriate abstraction (the domain or lattice): one that is simple enough for efficient computation, but expressive enough to retain precision. There is no systematic way to choose the abstraction. An analysis designer uses intuition and their experience, and often needs to discard an attempt and start over.

Consider the following code.

```
x = 0;
y = read_even_integer();   // produces an even value
x = y+1;
y = 2*x;
x = y-2;
y = x/2;
```

Assume all values are integers and there is no overflow.

It will be simpler to consider this code snippet in *static single assignment (SSA) form*, in which every variable is assigned on only one line of the program, and no variable is ever reassigned.

Here is the program and its SSA version:

```
x = 0;                  x1 = 0;
y = read_even();        y1 = read_even();
x = y + 1;              x2 = y1 + 1;
y = 2 * x;              y2 = 2 * x2;
x = y - 2;              x3 = y2 - 2;
y = x / 2;              y3 = x3 / 2;
```

Using SSA form resolves questions of scoping and which definition of a variable should be used. A special ϕ function is used at join points. For more details, see `https://en.wikipedia.org/wiki/Static_single_assignment_form`.

Exercise 3 *What is the most precise possible information about* x *and* y *(in SSA form, about* x3 *and* y3*) at the end of execution? You may wish to express your answer in terms of other variables, such as x_1, y_1, etc. (Assume ideal execution: all operations are exact, there is no overflow, etc.)*

Now, reflect on how you determined this information. However you determined it, a computer can do the same. An effective way to produce a program analysis is to observe how you deduced a fact, and then automate your mental process.

In this case, you probably performed "symbolic execution": for each variable value, determine an algebraic formula that represents its value, and then substitute that formula for each use.

Another fact that is true is that both x and y are even. Surprisingly, the even/odd abstract interpretation defined in section 2.2.2 cannot establish this fact!

Exercise 4 *Evaluate the code given in section 2.5 using the even/odd abstract interpretation. What is the final estimate for the values of* x *and* y*?*

Exercise 5 *Give an abstract domain that can be used to determine that the final value for* y *is even. The abstract domain only has to work for this particular program and property. Also give the transfer functions for* +*,* **,* -*, and* /*.*

2.6 The dataflow analysis algorithm

2.6.1 Terminology

Before giving an algorithm for performing abstract interpretation, we need to define some terms.

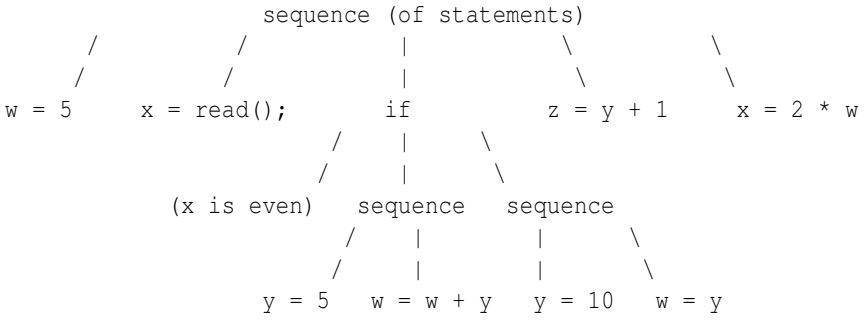
basic block A basic block is a sequence of contiguous statements that are always executed together. There are no jumps/gotos out of a basic block except at its end. There are no jumps/gotos into a basic block except at its beginning. For more details, see `https://en.wikipedia.org/wiki/Basic_block`.

CFG, control flow graph The control flow graph represents each basic block (and often each conditional expression) as a node. There is an edge between statement s_1 and statement s_2 if execution of statement s_1 can be immediately followed by s_2. A goto statement induces an edge. In an if statement, there are two edges from a conditional expression, and there are two edges into the statement that follows the if statement (the latter is called a join point. There is an edge from the end of a loop to the boolean expression that controls loop iteration. The control flow graph is the most convenient program representation for static analysis. For more details, see `https://en.wikipedia.org/wiki/Control_flow_graph`.

As an example of a control flow graph, consider this code:

```
w = 5
x = read()
if (x is even)
  y = 5
  w = w + y
else
  y = 10
  w = y
z = y + 1
x = 2 * w
```

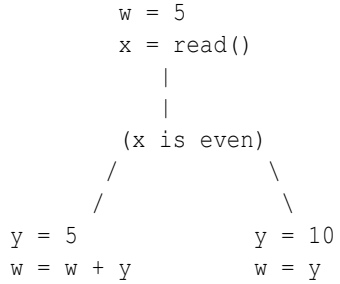
It contains 5 top-level statements, the third of which is an if statement. Its parse tree is:

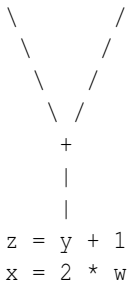


The parse tree shows the syntactic structure of the code, but does not make explicit the flows of control in the program.

(Typically, rather than the parse tree, we use the simpler abstract syntax tree (AST). The AST is very similar to the parse tree, but it elides some details. For example, parentheses are not needed because precedence is implicitly represented in the tree structure. For more details, see `https://en.wikipedia.org/wiki/Abstract_syntax_tree`.)

The control flow graph of this code is





The "+" represents a *joint point*, which has multiple predecessors — multiple statements could be the one that executes immediately before this program point.

2.6.2 Algorithm

To analyze a program with a dataflow analysis, first convert the program into a control-flow graph, use the following fixed-point worklist algorithm.

1. Start with an initial estimate of the abstract store on every control flow edge, including before every entry point (e.g., the start of the program) and at every exit point. This estimate typically maps every variable to $\bot$.

2. Put each of these program points in a worklist.

3. Until the worklist is empty, choose an arbitrary program point (= control flow graph edge) from the worklist.

   If this edge leads into a basic block, abstractly execute that basic block; that is, apply the transfer functions to determine the abstract store after the basic block.

   If this edge leads into a join point, use the lub function to determine the abstract store after the join point.

   For each exit edge of the basic block or join point: if the new output store differs from the abstract store that currently exists on that exit edge, replace the existing store and add the edge to the worklist. If this makes no change to the abstract store, add nothing to the worklist.

This is called a fixed-point algorithm because it continues doing work until doing work would have no effect. Section 2.15 further explains why this is called a "fixed-point algorithm".

This algorithm is an "optimistic algorithm". That means that it starts with a property that is strictly stronger than the final value, and the estimates become progressively weaker until a fixed point is reached. If you were to stop the algorithm early, then the results would be unsound. By contrast, a "pessimistic algorithm" starts with a weak property (typically, the abstract store maps every variable to $\top$) and iteratively strengthens the estimates until a fixed point is reached. An optimistic algorithm finds the least (most specific, most informative) upper bound, but you cannot peek at the answer until it finishes. A pessimistic algorithm finds the greatest upper bound, but you can stop it whenever you like (such as when you run out of time) and safely use the current estimate.

2.7 If statements, joins, and the least upper bound operation

When an if statement runs in a normal execution, either the then-branch or the else-branch is executed, but not both. By contrast, static analysis symbolically executes both branches. More specifically, if the estimate for the boolean value of the if-condition is $\{\text{true}, \text{false}\}$, then the abstract interpretation evaluates both the consequent and the alternative. These two threads of control re-join at a join point after the if-statement.

After an if statement, a variable might have a value that was assigned in the then-branch or in the else-branch. The estimate after the if statement needs to be at least as general (that is, at least as high in the lattice) as in each of the branches. This is exactly what the least upper bound operation (lub) does. For every variable x, post-if-store$(x) =$ post-then-store$(x) \sqcup$ post-else-store(x).

2.8 Loops

No special syntax is needed for loops, because a loop can be expressed as an if statement plus a goto (a back-edge) in the CFG (control flow graph).

However, in the presence of back-edges, the analysis described so far might never terminate! It might try to execute infinitely many different paths: all those that are possible at run time.

More concretely, suppose that a program contains a loop that can be entered. In other words, at run time it is possible for the loop condition to be true. If the abstract interpretation is sound, then the abstract value for the loop condition is either { true } or { true, false }. When abstract interpretation reaches the loop condition, it will enter the loop body. When the abstract interpretation reaches the end of the loop, it will take the back-edge, proceed to the loop condition, and again execute the loop body. Since there are infinitely many looping paths through the program, it seems that abstract interpretation might never terminate.

In practice, an abstract interpretation halts because it performs a fixed-point analysis as follows:

- Whenever a new abstract value flows along a control flow edge, re-execute the basic block(s) that it leads to. (This produces an abstract store for its output edge.)
- If the output store of a basic block is the same as it was the last time, then terminate the analysis along that path; that is, don't continue to the successor basic block. There might be other statements waiting to be analyzed however.

Consider the following code that multiplies x and y:

```
x = read_positive()
y = read_positive()
result = 0
loop:
result = result + y
x = x - 1
if x != 0 goto loop
```

Let's analyze this code using several different abstract interpretations. For each exercise, give the final store — that is, what properties are known to be true for each variable after the loop.

Exercise 6 *The "even, odd, unknown" abstraction.*

Exercise 7 *Use the same abstraction, but analyze the loop 4 times, starting from each of these 4 possibilities at the* beginning *of the loop:*

1. $[\mathsf{x} \to \mathsf{even}, \mathsf{y} \to \mathsf{even}, \mathsf{result} \to \mathsf{even}]$
2. $[\mathsf{x} \to \mathsf{odd}, \mathsf{y} \to \mathsf{even}, \mathsf{result} \to \mathsf{even}]$
3. $[\mathsf{x} \to \mathsf{even}, \mathsf{y} \to \mathsf{odd}, \mathsf{result} \to \mathsf{even}]$
4. $[\mathsf{x} \to \mathsf{odd}, \mathsf{y} \to \mathsf{odd}, \mathsf{result} \to \mathsf{even}]$

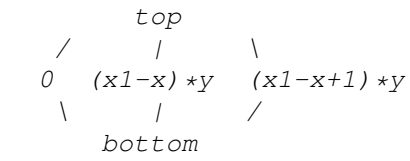
Exercise 8 *In the evenness analysis as presented so far, the abstract store or abstract state is a mapping from variables to abstract values, such as* $[\mathsf{x} \to \ldots, \mathsf{y} \to \ldots, \mathsf{result} \to \ldots]$. *For the variables* x, y, *and* result, *you can think of this as a triple of abstract values.*

Consider a different, more expressive abstract domain that is a set of triples of abstract values. For example, the abstract state $\{\langle \mathsf{even}, \mathsf{odd}, \mathsf{odd}\rangle, \langle \mathsf{odd}, \mathsf{even}, \mathsf{odd}\rangle\}$ *means that it is possible for* x *to be* even, y *to be* odd, *and* result *to be* odd, *and it is also possible for means that it is possible for* x *to be* odd, y *to be* even, *and* result *to be* odd. *The only way to express such possibilities in a single abstract store is as* $[\mathsf{x} \to \mathsf{unknown}, \mathsf{y} \to \mathsf{unknown}, \mathsf{result} \to \mathsf{odd}]$, *which is less informative.*

The exercise is to formalize this abstract interpretation by stating its lattice, lub, and transfer functions. Is an analysis using this abstract interpretation guaranteed to terminate?

Exercise 9 *Try an analysis whose domain is symbolic expressions for the original code. What is a problem with this domain?*

Exercise 10 *Try this domain:*



where x1 *means the initial value of* x *assigned by* read_positive(). *This analysis gives a very precise result, but the domain is very peculiar and seems to have been pulled from a hat. How could a programmer or tool have come up with this domain?*

2.9 Termination

Section 2.6.2 gave a rule that stops analysis of a basic block if its (abstract) inputs have not changed since the last time the block was analyzed. Given that rule, an abstract interpretation is guaranteed to terminate! Using the standard abstract store that maps each variable to an abstract value, a loop is analyzed at most *numberOfVariables* × *heightOfLattice* times. (If there are infinite ascending chains, then *heightOfLattice* is infinite and the analysis may never terminate. See section 2.17 which discusses a technique called widening.)

Exercise 11 *Why is a loop is analyzed at most numberOfVariables* × *heightOfLattice times?*

Note that infinite/long loops in the program being analyzed and infinite/long loops in the analysis are orthogonal. Neither implies the other.

Exercise 12 *Consider an abstract domain containing these abstract values. Their meanings are given to the right of their names.*

odd	x % 2 == 1
even	x % 2 == 0
mod4	x % 4 == 0
is2	x == 2

Give the lattice for such an abstract domain.

Exercise 13 *Write out the transfer function for* +, *using the lattice of exercise 12.*

Exercise 14 *Analyze this procedure using the lattice of exercise 12 and transfer function of exercise 13.*

```
def g() {
  x = 2;
  while (x < 10) {
    x = x + 2;
  }
  return x;
}
```

Note that even though g *may execute many iterations of the loop, the abstract interpretation completes quickly.*

Exercise 15 *Consider the following program, which adds* x *and* y*:*

```
sum = 0
loop:
if x > 0
  sum = sum + 1
  x = x - 1
  goto loop
```

Analyze this program using an abstract domain whose values are var $\leq$ constant. For example, two of the abstract values are $x \leq 4$ *and* $x \leq 5$*. Note that this domain has infinite height; is that a problem?*

Exercise 16 *Analyze the program of exercise 15, using an abstract domain whose values are var $\geq$ constant. For example, two of the abstract values are* $x \geq 4$ *and* $x \geq 5$*. Note that this domain has infinite height; is that a problem?*

2.10 Monotonicity

Here are two functions with the same type signature, that are both used during abstract interpretation:

$$\begin{aligned} lub &: T \times T \to T \\ transfer[+] &: T \times T \to T \end{aligned}$$

These are completely unrelated functions, used for different purposes. Don't confuse them.

Both functions must be monotonic. A binary function f is monotonic if $\forall a,b,c,d.a \sqsubseteq b \wedge c \sqsubseteq d \Rightarrow f(a,c) \sqsubseteq f(b,d)$.

A property that is frequently confused for monotonicity is: $\forall x.f(x) \sqsupseteq x$. (For a binary function, $\forall x,y.f(x,y) \sqsupseteq x \wedge f(x,y) \sqsupseteq y$.) Monotonicity does not imply this property, and this property does not imply monotonicity. This property holds of the *lub* function, but this property need not hold of transfer functions. For example, given the even–odd domain, the transfer function for x++ can change the estimate for x from even to odd, but even and odd are unrelated in the lattice.

Exercise 17 *What might go wrong if lub or a transfer function is not monotonic? Give an example of a lub function that is not monotonic and that causes the problem. Using the same lattice and a lub function that is monotonic, show that the problem does not occur.*

The sequence of estimates that is the input or output to a statement's or basic block's transfer function is monotonically increasing.

Exercise 18 *Why? How is this important in the claim of termination?*

2.11 What is Top?

English is imprecise. People sometimes speak of $\top$ as representing "no information", because nothing is known about the possible run-time values. People sometimes speak of $\top$ as representing "all information", because every value is possible. The former statement is a better way to think about $\top$, but in any event you should avoid ambiguous statements that can be misinterpreted.

$\top$ represents *no* constraint on the possible values. The set of values represented by $\top$ includes *every* possible value. Both are valid ways of thinking about $\top$, but each time you think about an analysis, choose one of them and stick with it.

Likewise, $\bot$ represents every possible constraint on the values — so many constraints that they are unsatisfiable. The set of values represented by $\bot$ is the empty set.

2.11.1 Transfer function results for Top and Bottom

When $\top$ represents "arbitrary run-time value", then often but not always, an operation one of whose arguments is $\top$ will yield $\top$.

When $\bot$ represents "no possible run-time value", then usually an operation one of whose arguments is $\bot$ will yield $\bot$. It would be correct, but less precise, to return a different value.

There are some exceptions. For example, the abstract value for `x` after `x=...` is the same no matter the abstract value for `x` immediately before the operation. As another example, in the even/odd abstract domain, `2*x` evaluates to an even value if `x` is top, but to bottom if `x` is bottom.

2.12 Examples

This section illustrates a variety of useful abstract interpretations.

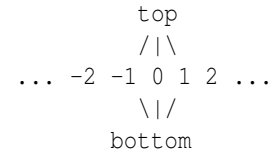
Here is a program to use as a test case for all the following example analyses.

```
w = 5
x = read()
if (x is even)
  y = 5
  w = w + y
else
  y = 10
  w = y
z = y + 1
x = 2 * w
```

2.12.1 Constant propagation

Goal: for each variable, determine if its run-time value can be computed at compile time. If so, then other analyses can take advantage of that information, and compilers can replace uses of the variable by its value.

Lattice for a single variable:



Let b be a basic block and f_b be the transfer function for the basic block. Let $constants_{pre}(b)$ be the store at the beginning of the basic block, and similarly for $constants_{post}$.

For any basic block b,

$$\begin{aligned} constants_{post}(b) &= f_b(constants_{pre}(b)) \\ constants_{pre}(b) &= \bigsqcup_{p \in pred(b)} constants_{post}(p) \end{aligned}$$

2.12.2 Copy propagation

Goal: Given a variable `x`, what other variables are equal to `x`? This is valuable for compilers because if there are any, then `x` can be optimized away. Analogous to constant propagation, the compiler can eliminate variables.

The domain is sets of equal variables. For instance, one domain element is

$$\{\{a\},\{b\},\{c,d\},\{e,f,g\}\} .$$

Each variable is known to be equal to the other variables in its set. It might also be equal to other variables, since the abstraction is an approximation to the actual equality relationships.

So far, in all our examples the lub and transfer functions have been defined over single variables, and they are extended to full domain elements (which give a value to every variable in the program) pointwise. For copy propagation, this approach does not work, and the functions are a bit more complicated to define.

Exercise 19 *Define an abstract interpretation for copy propagation.*

2.12.3 Live variables

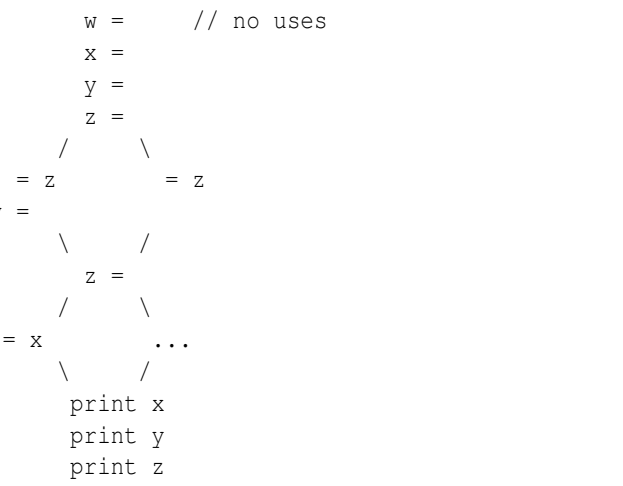
A "dead variable" is one whose value will not be used in the future. A "live variable" is one whose value might be used in the future. In other words, a dataflow analysis has two possible responses to the question, "Will this variable be used in the future?": "no" or "maybe". A sound analysis is allowed to answer "maybe" even if the true answer is "no", but it is not allowed to answer "no" if there is any possible execution where the variable is used in the future.

Most compilers perform this analysis to perform optimizations such as reusing a register, deallocating a value, etc.
A general approach for creating an analysis is

1. Determine the answer manually.

2. Review your reasoning process: what information did you use, and what reasoning steps did you perform? Formalize and automate your manual work.

Let's go through that process. Below is the control flow graph (CFG) for a program. (Irrelevant left-hand-sides asd right-hand sides of assignments have been elided to avoid clutter.) Before proceeding, manually determine, at each program point (that is, before and after each line of code), which of the 4 variables `w`, `x`, `y`, and `z` is live at the program point. For example, the answer at the beginning of the final basic block is {`x`, `y`, `z`} because all three variables will be used in the future, but `w` will not be used in the future. The answer at the beginning of the first basic block is {} because the current values of all the variables will never be read — each variable is reset (assigned to) before it is used.



Now that you have manually determined the answer, you can choose an abstract domain and define transfer rules.

The key observation you should have made is that computing live variables is a *backwards analysis*. A regular, forwards analysis gives facts about the execution so far. A backwards analysis gives information about the future execution.

Transfer function for live variables

The lattice points are the elements of the powerset of {`w`, `x`, `y`, `z`}.

Let s represent the set of live variables. Then the transfer function for the statement `a=b` is *almost*

$$s_{pre} = s_{post} - \{a\} \cup \{b\} \ .$$

(It is a convention to use primed variables to mean "post-state" and unprimed variables to mean "pre-state", so an alternative formulation of the rule would be $s = s' - \{a\} \cup \{b\}$. That convention is a bit obscure, however.)

To see why that rule does not work, consider the statement `a=a+1`, in which a variable appears on both the lhs and the rhs of the assignment. Variable `a` should be live before the assignment iff it is live after the assignment.

We often write transfer functions for entire basic blocks rather than statement-by-statement, and such transfer functions very often have a variable in both the kill-set and the gen-set. (Long-standing terminology uses "kill" for "remove from the set" and "gen" for "add to the set"; many dataflow analyses are so-called "gen–kill analyses".)

Here is a definition of the transfer functions:

$$\begin{aligned} use(p) &= \text{may be used } \textit{before being defined/set} \\ def(p) &= \text{must be defined} \\ live_{out}(v) &= \bigcup_{s \in succ(v)} live_{in}(s) \\ live_{in}(v) &= use(v) \cup (live_{out}(v) - def(v)) \end{aligned}$$

Try it on this example:

```
x = read();
y = 10;
z = x+1;
w = y*x;
y = 5*z;
x = 12;
```

2.13 Procedure calls

There are two main approaches to analyzing a program that contains procedures.

Inlining At each call site, inline the procedure body. Each instance of the procedure body is analyzed separately, which increases precision because different facts may be true at different call sites.

If the program contains the possibility of recursion, this approach does not work because the inlining never stops. Even in the absence of recursion, this approach can exponentially increase the size of the analyzed program and the cost of the analysis.

Summarization Analyze each procedure body once and create a summary of the procedure's effect. You can think of this summary as the transfer function for a call to the procedure. Now, at each call site, use the summary.

2.14 Richer transfer functions: refinement and branching

A simple way to express a transfer function, such as that for `+`, is as taking two arguments (each an abstract value) and producing one result (an abstract value). The language of section 2.2.2 lets us write `+` only in the context of an assignment, so our abstract interpretation has a transfer rule not for `x + y` but for `z = x + y`.

Consider the problem of verifying that a program suffers no null pointer exceptions. Think about how you would analyze this program:

```
if (z != null) {
  z.f
}
```

You should have reasoned that z is non-null within the then branch. Such reasoning is beyond the capabilities of how we have expressed transfer functions so far.

So far, each transfer function has been for an operation that creates a value, such as + or *. This reflects the fact that when a variable changes, the abstract interpretation's estimate of the variable changes: the estimate is now based on the new value rather than the old value. The new estimate can be higher, lower, or unrelated to the old estimate.

There is another way that an abstract interpretation can update its knowledge about a variable: via a test in the source code. The new estimate is always lower (more informative) than the old estimate, which is why this process is called "refinement".

To obtain information from operations, such as if tests and assert statements, we will make two extensions:

refinement A transfer function can determine a new abstract value for any variable, not just those on the left-hand side of an assignment. It might change (refine) the abstract value of variables that are passed as arguments or even ones that do not appear in the given line of code.

branching A transfer function for a boolean-valued expression has two different outputs, each an abstract store. If the expression is used in an if statement, then each output is used on a different branch. If the expression is not the predicate in an if statement, then the two abstract values are lubbed and that single result is used.

Consider the following program, analyzed using the even–odd domain.

```
                                  w     x     y
                                  T     T     T
y = x * 2
                                  T     T     E
if (w % 2 == 0) {
                                  E     T     T
  print "even"
                                  E     T     T
} else {
                                  O     T     T
  print "odd"
                                  O     T     T
}
                                  T     T     E
if (w % 4 == 0) {
                                  E     T     T
  print "multiple of 4"
                                  E     T     T
} else {
                                  T     T     T
  print "not multiple of 4"
                                  T     T     T
}
                                  T     T     E
assert x % 2 == 0
                                  T     E     E
```

The first statement illustrates updating the abstract store based on an assignment.

After the w % 2 == 0 test, on the then branch w has the abstract value odd, and on the else branch w has the abstract value even. The abstract store after the if statement is the lub of the two branches; since no assignments occurred within the if statement, this is the same as the abstract store before the if statement.

The second if statement illustrates that the else branch is not always updated in the opposite way as the then branch. In the second if statement, refinement occurs in the then branch but not in the else branch.

Finally, an assert statement also gives information: if execution proceeds past the assert statement, then x is even.

Returning to the null pointer example, note that these extensions can handle

```
if (z != null) {
  z.f
}
```

but not

```
p = z != null;
if (p) {
  z.f
}
```

and not

```
if (p) {
  z = some-non-null-value
}
if (p) {
  z.f
}
```

Exercise 20 *How could you handle those cases? What are the pros and cons of such extensions to abstract interpretation?*

Exercise 21 *Define an analysis that computes, for every program point, which variables contain a value that has definitely been used by the program up to that point. (This is a form of strictness analysis.)*

Then, run the analysis on this program, filling in each blank with the abstract store at the given program point. Recall that in the usual case, the abstract store is a mapping from a variable to an abstract value; use that representation for this exercise.

```
def f(a, b, c) {
                    [ a -> _____, b -> _____, c -> _____, d -> _____ ]
  if (*) {
                    [ a -> _____, b -> _____, c -> _____, d -> _____ ]
    d = a + b;
                    [ a -> _____, b -> _____, c -> _____, d -> _____ ]
  } else {
                    [ a -> _____, b -> _____, c -> _____, d -> _____ ]
    d = a + c;
                    [ a -> _____, b -> _____, c -> _____, d -> _____ ]
  }
                    [ a -> _____, b -> _____, c -> _____, d -> _____ ]
  return d;
                    [ a -> _____, b -> _____, c -> _____, d -> _____ ]
}
```

*(Note: the * condition means "random choice") Hint: you will need to define a lattice and define transfer functions for* x = y + z *and* return x.

2.15 Fixed points

Abstract interpretation is a type of fixed point analysis.

A *fixed point* of a function $f : T -> T$ is a value $x : T$ such that $f(x) = x$. For instance, 0 and 1 are fixed points of the sqrt function.

An abstract interpretation computes an abstract state at every program point. In the simplest case, the abstract state of a program is a mapping from each variable to an abstract value. Other times, the abstract state has a different form, as in the copy propagation analysis of section 2.12.2.

The algorithm given in section 2.8 examines one statement at a time and, if that statement's post-state changes, proceeds to the statement's successors. It is a dataflow algorithm that takes a local view, which is how dataflow analysis is implemented in practice.

Another perspective is of the symbolic execution proceeding in parallel for every statement in the entire program. That is, the symbolic execution takes as input a collection of abstract states (one per program point) and produces as output a set of abstract states (one per program point), by locally applying the transfer function once. The result of the program analysis is the fixed point of that big composed function. When the program analysis terminates, the current abstract state is the fixed point of the big composed function.

It is also possible to view this more locally, at a single program point. Consider all paths from the program point back to the program point. Imagine a transfer function that represents the effect of all those paths. This transfer function has a fixed point, and the abstract interpretation computes that fixed point.

Using the big composed function requires an initial state everywhere in the program. (The implementation only needs an initial state at entry points, such as the beginning of main and any other publicly-visible procedures.) This initial state is the bottom state. Whenever a transfer function is applied, estimates *only go up and never go down* in the lattice. The estimates go up until they reach a fixed point, and then they stop increasing. This is why abstract interpretation is guaranteed to reach the least (that is, most constrained and informative) fixed point, even if other fixed points exist.

2.15.1 Estimates increase

An inductive proof shows that the estimate always goes up — that is, whenever an abstract value changes, it changes in the upward direction.

- The base case is the initial state, which is bottom. Every non-bottom value is above bottom in the hierarchy.
- The inductive case has two subparts: for transfer functions and for lub at joint points. When new information is produced by a transfer function (because the transfer function is run on different (higher!) abstract inputs than it observed before), the output is at least as great as any previous output. This is because a transfer function f must be monotonic: if $a \sqsubseteq b$, then $f(a) \sqsubseteq f(b)$. (However, it is possible that $f(a) \sqsubset a$.) The lub function is also monotonic, completing the proof.

Intuitively, an abstract value is an approximation of all values that can reach a program point, and as the analysis proceeds, it can only learn about more values that can reach the program point. Recall from section 2.2 that higher points in the lattice represent larger sets of possible values.

2.16 May vs. must analyses

A "may" analysis determines whether a fact is true on *some* path through the program. A "must" analysis determines whether a fact is true on *all* paths through the program.

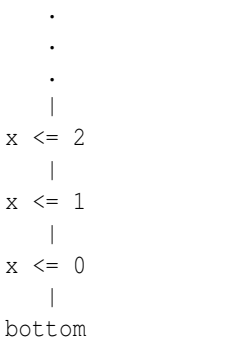
There is a very simple way to convert a may-analysis to a must-analysis or vice versa: just turn the lattice upside down!

Another way to view this is to use glb (greatest lower bound) rather than lub at join points.

Exercise 22 *Repeat exercise 21, but as a may-analysis. That is, determine which variables contain a value that* may *have been used by the program up to that point.*

2.17 Widening

Consider the following lattice:



Using that lattice, symbolically execute this loop:

```
x = 0
while !(x == answer) {
  x++
}
```

The symbolic execution never ends: the estimate for x just keeps getting higher and higher.

For an abstract analysis to terminate, the lattice must have no infinite ascending chains. Finite but long ascending chains (such as the above lattice limited to 32-bit machine integers, where the top element is $x \leq 2147483647$) lead to impractically long analysis times.

An abstract interpretation can handle both situations via a sound heuristic called "widening".

If a particular statement in the program is encountered too many times during an abstract interpretation, then perhaps the abstract interpretation is stuck in an infinite or very long fixed-point loop. In that case, lub operator intentionally returns an abstract value that is higher than the most precise result.

Here is an example. Suppose a loop has been analyzed 10 times. On the 11th iteration the analysis would use the following lub operator:

$$lub(x \leq k, x \leq l) = \begin{cases} x \leq k & \text{if } k = l \quad \text{(This abstract value didn't change on this iteration.)} \\ x \leq 2^7 - 1 & \text{if } k < 2^7 \wedge l < 2^7 \\ x \leq 2^8 & \text{if } k \leq 2^8 \wedge l \leq 2^8 \\ x \leq 2^{15} - 1 & \text{if } k < 2^{15} \wedge l < 2^{15} \\ x \leq 2^{16} & \text{if } k \leq 2^{16} \wedge l \leq 2^{16} \\ x \leq 2^{31} - 1 & \text{otherwise} \quad (\text{“}x \leq 2^{31} - 1\text{” is } \top.) \end{cases}$$

The rationale for this lub function is that the value might take on all values that fit within a certain programming language type: signed or unsigned byte, short, or int. A more sophisticated lub operator might also use constants that appear in the loop being analyzed.

This widening heuristic is intended to quickly skip over many states in order to get more quickly to a fixed point. It may skip over the least fixed point.

Exercise 23 *Why is the widening heuristic sound?*

2.18 Tightness of an analysis

An abstract interpretation gives an estimate of what might occur at run time. This estimate may be precise — that is, close to the truth — or it might be a wild overestimate. Here, "the truth" means the actual possible behaviors; the truth

is generally unknowable because of the halting problem. An analysis might produce a precise estimate for one program and a loose estimate for another program. There is no way to know *a priori* how accurate it will be.

One way to bound the truth — and incidentally to estimate the analysis's accuracy — is to write two analyses:

- a sound, conservative analysis that upper-bounds what can happen at run time
- an unsound, optimistic analysis that lower-bounds what can happen at run time. (Running a test suite is one example of a lower-bound analysis!)

You know that the answer is between the two bounds. If the bounds are near one another, you know that the truth is near both of them: both estimates are close to the truth. If the bounds are far apart, then at least one estimate is far from the truth, but you don't know which one or where the truth lies. Surprisingly often, a dynamic analysis run with a non-trivial test suite gives a closer estimate of the truth than a sound analysis.

2.19 Tip: Make your abstract domain as simple as possible

Suppose that you want to prove that a program has no null pointer exceptions — or, equivalently, to identify all the places that a null pointer exception could occur.

Here is a possible lattice for a nullness analysis:

```
     Top = unknown = {null, non-null}
    /   \
{null} {non-null}
    \   /
   Bottom = no possible values = {}
```

However, there is no need for {null} or $\bot$. The analysis will issue a warning anytime a value is possibly null, so it treats $\top$ the same as {null} and $\bot$ the same as {non-null}: the distinction between $\top$ and {non-null} is the only interesting distinction. (As a minor point, other than the null literal itself, programmers rarely write expressions with the type {null}, meaning that the expression always evaluates to null on every execution.)

2.20 Relationship between concrete and abstract executions

The abstraction function α maps from a concrete value to an abstract value. For the even/odd domain, $\alpha(4) = \mathsf{even}$.

The concretization function γ maps from an abstract value to a set of concrete values. For the even/odd domain, $\gamma(\mathsf{even}) = \{\ldots, -2, 0, 2, \ldots\}$.

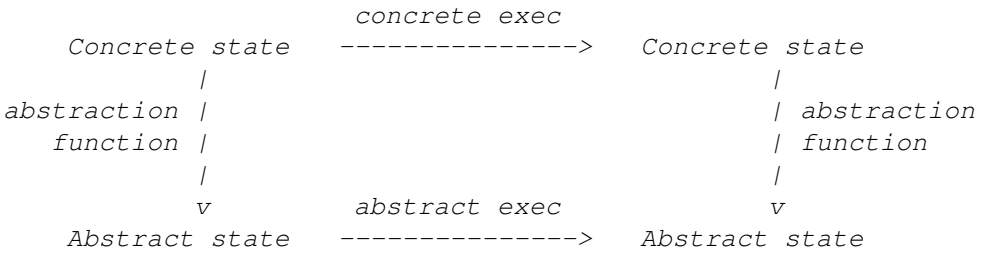
It is a requirement that for all x, $x \in \gamma(\alpha(x))$. Usually, $\gamma(\alpha(x))$ contains many more values than x. Treating the concrete domain as $\mathcal{P}(\mathbb{N})$, the powerset of the natural numbers, this is written as $x \sqsubseteq \gamma(\alpha(x))$. Satisfying this property means that the abstract and concrete lattices are in a relationship called a "Galois connection".

Here is a visual representation of the relationship between concrete and abstract execution:

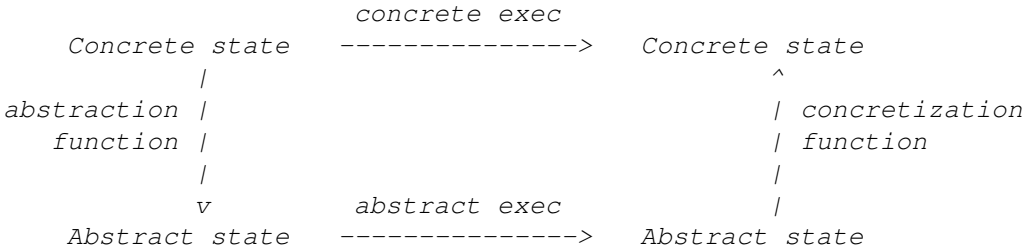
```
                    concrete exec
    Concrete state  --------------->  Concrete state
           |
abstraction |
   function |
           |
           v        abstract exec
    Abstract state  --------------->  Abstract state
```

Here are two questions you can ask about the diagram, for a given abstract interpretation.

Exercise 24 *What is the relationship between the abstract states that the following two paths lead to? For example, do the two paths lead to the same abstract state?*



Exercise 25 *What is the relationship between the concrete states that the following two paths lead to? For example, do the two paths lead to the same concrete state?*



2.21 More exercises

Exercise 26 *Section 2.2.1 gave three properties of a well-formed lattice. For each of the properties, what could go wrong if it is violated? Also, why are infinite descending chains permitted?*

2.21.1 Reasoning about GCD

Stein's algorithm (`https://en.wikipedia.org/wiki/Binary_GCD_algorithm`) computes the greatest common divisor (GCD) of two integers. On some CPU architectures, Stein's algorithm admits a faster implementation than the far more famous Euclidean algorithm (`https://en.wikipedia.org/wiki/Euclidean_algorithm`).

Here is one possible implementation of Stein's algorithm:

```
def gcd(int a, int b):
    if a == 0 or b == 0:
        return 0
    int expt = 0
    while a is even and b is even:
        a = a / 2
        b = b / 2
        expt = expt + 1
    while a != b:
        if a is even:    a = a / 2
        elif b is even:  b = b / 2
        elif a > b:      a = (a - b) / 2
        else:            b = (b - a) / 2
    return a * 2^expt
```

Exercise 27 *A program will crash if it ever divides by zero. Does this program ever crash due to division by zero? Carefully explain how you figured this out. Explain briefly how an abstract interpretation could determine this fact.*

Exercise 28 *Stein's algorithm is intended to compute a value; it should always terminate. Is it possible for this program to run forever? Explain your reasoning. Explain briefly how an abstract interpretation could determine this fact.*

2.21.2 Division by zero

Divide-by-zero errors often catch developers off guard[1]. They can happen due to division by zero (e.g., $1/0$) or mod by zero (e.g., $1\%0$).

Exercise 29 *Give an abstract interpretation that catches possible divide-by-zero errors statically. You do not need to give transfer functions for every integer operation; addition, multiplication, and division will suffice. Think carefully about your choices; the design space is quite large!*

You may assume that (1) we only care about integers (not floating-point values), and (2) overflow does not happen.

Briefly contrast your design to an alternative you chose. Name one way in which your design is better than the alternative, and one way in which it is worse.

Exercise 30 *Implement your design (or a simplified version of it, as stated above) for integer division in Java. You can find skeleton code and detailed instructions here:* `https://github.com/kelloggm/div-by-zero-checker/blob/master/INSTRUCTIONS.md`.

Please fork this repository, and submit a link to your forked repository, which contains your implemented analysis.

2.22 Notes on Cousot & Cousot POPL 1977 paper

Cousot & Cousot's POPL 1977 paper [3] is the classic citation about abstract interpretation. That paper provides a mathematical explanation of dataflow analysis, which was already widely used at the time.

The paper uses a number of terms and notations without defining them. Here is some information to help you make sense of the paper.

Declaration notation $f : T|P$ "$f : T|P$" defines f to be a thing of type T that satisfies property P.

Semilattice A lattice with a well-defined lub, but not necessarily a well-defined glb (or vice-versa). You will sometimes see "join semilattice" for one with only a lub operation, or "meet semilattice" for one with only a glb operation.

Complete lattice or complete semilattice The formal definitions are quite involved, but the key fact about complete lattices is that they are bounded — they have top and bottom elements and a finite height.

Order-preserving, isotone, monotone, monotonic A function f is order-preserving if $x \leq y$ implies $f(x) \leq f(y)$) "Isotone", "monotone", and "monotonic" are other words for order-preserving.

Fixpoint A value x associated with a function f such that $f(x) = x$. If f is order-preserving with respect to a lattice, then repeatedly applying f (see Kleene's sequence immediately below) will eventually arrive at some fixpoint.

Kleene's sequence The Kleene's sequence k for a function f is

$$\begin{aligned} k_0 &= \text{some initial value} \\ k_i &= f(k_{i-1}) \end{aligned}$$

A finite Kleene's sequence is one where the value eventually stops changing; i.e., there exists j such that $k_i = k_j$ for all $i > j$. The value k_j is called the "limit" of the sequence, and it is a fixpoint of f.

Semantics If you have not seen formal semantics for a language before, Section 3.2 will be quite dense. The authors are simply defining an interpreter in terms of mathematical constructs.

[1] `https://github.com/search?q=divide+by+zero&type=Issues`

2.23 Notes on "Abstract Interpretation: a semantics-based tool for program analysis" by Jones & Nielson

This paper claims to start out informally, but it dives into mathematical notation fast. You don't need most of the "desirable mathematical background" for the first sections.

The main focus of the first sections is the relationship between the concrete and abstract domains, which is discussed in section 2.20 of this book.

You can think of a *complete partial order* as a lattice (see definition in section 2.2.1), though actually a lattice has some additional properties.

A *homomorphism* is a mapping between two datatypes that preserves operations. For example, suppose h is a mapping between datatype A and datatype B, both of which have a $\cdot$ operation. The $\cdot$ operations have corresponding specifications, but probably different implementations, in the two datatypes. Then for all $x, y \in A$, $f(x \cdot y) = f(x) \cdot f(y)$.

In abstract interpretation, we are particularly interesting in the mapping between the concrete domain and the abstract domain. However, the equality generally does not hold. Exercises 24 and 25 demonstrate this. The relation is $\sqsubseteq$ rather than $=$: "The effect is that the price paid for exact computability is a loss of precision." Another way of saying that equality does not hold is that the diagram in exercise 24 does not "commute": traversing different paths to the same endpoint does not yield the exact same abstract state.

What Jones & Nielson call a "representation function" (β), this book calls an "abstraction function" (section 2.4).

2.24 Notes on "Static and dynamic analysis: synergy and duality"

This paper is somewhat difficult to read for an outsider, for two reasons. First, it is aimed at motivating the program analysis research community, not at introducing outsiders to a field. Second, it had to fit in a 4-page limit (including references). Both of these factors lead it to use some terminology, and to omit some explanations, that would be useful to a reader.

The paper's structure is

2. Static and dynamic analysis: synergies
    (a) Performing both static and dynamic analysis
    (b) Inspiring analogous analyses
    (c) Hybrid static–dynamic analysis
3. Duality: subsets of behavior

However, it ought to have been structured this way, which is the structure of its argument:

2. Static and dynamic analysis: synergies
3. Inspiring analogous analyses
4. Hybrid static–dynamic analysis
    (a) Duality: subsets of behavior

The first part is about performing both static and dynamic analysis, that is, pre- and post-processing. There is nothing innovative about this: using distinct static and dynamic analyses to support one another was well-known in the community.

The second part, "inspiring analogous analyses", is the real contribution of the paper. It claims that any problem that can be solved statically can also be solved dynamically, and that any problem that can be solved dynamically can also be solved statically. Sometimes the algorithms are even the same. This was an observation that was not obvious to the research community at the time. It has led to significant amounts of impactful research. It may seem obvious in retrospect, but that is the mark of a good observation.

The third part, about combining static and dynamic analyses into a single analysis, is rather speculative. It notes that both static and dynamic analyses have access to only a subset of all the behaviors of a program — static analysis because of abstraction, and dynamic analysis because it observes only specific executions and records only parts of the full program state. Can these two views be unified, so that static and dynamic analysis can be explained in the same framework and more usefully compared? Can this lead to hybrid analyses that lie in between traditional static and dynamic analyses? Analyses of this type have not come to pass in the time since publication of the paper. Concolic execution (section 3.7) uses both dynamic and concrete analyses, though it switches between them, and uses each to guide the other, rather than hybridizing them as suggested in this paper.

The paper makes a questionable claim about turning a knob between soundness and precision, but isn't soundness an all-or-nothing proposition? In fact, many analyses give up soundness, such as ignoring hard-to-analyze features such as reflection, native code, and dynamic code loading, while still claiming to be sound.

Chapter 3

Testing

Testing is a dynamic analysis. A *dynamic analysis* runs the program and observes its operation. One example is profiling to determine which parts of a program consume the most run time or memory. Another example is parts of the debugging process.

Tasks related to testing include creating tests, prioritizing and selecting tests, and measuring test suite quality.

See appendix A.3 for definitions of the terms "defect", "error", and "failure".

3.1 The parts of a test

A test, or test case, consists of two parts: a test *input* and a test *oracle*.

The input is data upon which the software under test operates. The oracle is a predicate that determines whether the software under test has behaved correctly or incorrectly.

The weakest oracle is "the program runs". The weakest useful oracle is probably "the program does not crash". Though weak, it is better than nothing. Some automated "test generation" tools are actually test input generation tools, and they use this weak oracle.

3.2 Types of tests

Three types of tests are unit tests, integration tests, and system tests. (This book does not discuss other types, such as functional tests, smoke tests, stress tests, etc.)

A *unit test* executes just one part, or unit, of the software, without executing any other parts. Therefore, the test runs quickly, and if the test fails, you know exactly what part of the software is defective. A unit may be a procedure, class, module, or any other component.

Here is a unit test for procedure f:

```
a = 42;
b = g(...);
c = h(a, ...);
assert f(a, b) == c;
```

The *software under test (SUT)* is the procedure f. The inputs are the values of a and b. (The code in the test that computes these inputs is often called "setup code".) The oracle is the test "== c".

An *integration test* combines multiple units and ensures that they work when put together.

A *system test* executes a system as a whole, in the way it is expected to be used. The software under test is often a binary executable program. The input might be a file, file system, or stream of inputs. The oracle might be an expected file, file system, or stream of outputs.

3.3 Ways of generating tests

In order to write a test, a programmer or tester must create program inputs and oracles. There are two general approaches to this: black box testing and clear box testing. Each approach is applicable to most types of testing, including unit, integration, and system testing.

In *black box testing*, the test author relies on the specification of the software under test, such as its intended input–output behavior. The test author has no knowledge of the internals or implementation of the software; the software is treated as a "black box" that the test author cannot see inside.

In *clear box testing*, the test author reads the software's source code and uses that information when writing tests. Other names for clear box testing are *glass box testing* and *white box testing*. Clear box testing enables a test author to exercise a particular part of the software. For example, suppose that the software uses different algorithms in different situations, such as a list for small data structures and a hash table for large data structures. A test suite written using the clear box testing methodology would include tests that create data structures both below and above the size cutoff. Tests written using the black box methodology would not necessarily do so. Tests written using the clear box methodology can be written to achieve specific coverage guarantees. (See section 3.4 for a discussion of test coverage.)

Typically, a developer creates both types of tests: first using the black box methodology (ideally, after the software is specified but before it is implemented) and then using the clear box methodology (after the software is implemented).

All of the tests will pass for any correct implementation of the specification, but black box tests written for one implementation will not necessarily achieve coverage goals when run against a different implementation.

3.4 What can a dynamic analysis guarantee?

A sound static analysis, akin to a proof, can provide guarantees about every possible execution of a program. (Sometimes, though, its conservatism may lead it to produce such an imprecise answer that it is not useful.) By contrast, in general a dynamic analysis cannot provide the same guarantees: just because your program worked the first 10,000 times you ran it does not mean it will work the next time. As Edsger Dijkstra famously quipped, "program testing can be used very effectively to show the presence of bugs but never to show their absence." [4]

In theory, a dynamic analysis *can* give a guarantee; for example, testing can verify correctness. However, the dynamic analysis must be run on every possible input. In practice, this is impractical because it is too inefficient (or it is impossible, if the program's input domain is infinite). It's no worse (in terms of computability) than performing a perfectly precise static analysis that never performs abstraction nor throws away information. Both static and dynamic analysis are equally about deciding what information to abstract, which is an engineering tradeoff.

Although impractical, this complete approach should be in the back of your mind whenever your write a test suite. Your goal is to write a finite test suite that emulates the complete test suite in the sense of exercising many distinct behaviors of the software under test. Keep adding tests so long as they are not redundant with existing tests. (Or alternatively, think of writing the exhaustive version, but remove tests that are redundant.) Sometimes you can prove that two tests are redundant, but usually the redundancies are based on heuristics.

Here are some heuristics for deciding when tests are redundant:

partition the input Divide the input into subdomains, where you believe that the program behaves similarly (in terms of behaving correctly or incorrectly) on all the inputs in a given subdomain. You can use abstractions just as you might for static analysis. For instance, you might use the even/odd abstraction, and (for a routine with two inputs x and y) choose one input from each of the following subdomains:
x is even, y is even
x is even, y is odd
x is odd, y is even
x is odd, y is odd

corner cases Add inputs that cover corner cases where programmers often write defective code: 0, 1, -1, null, the empty list, an empty file, etc.. You can also use values that appear in the source code, in case there is an off-by-one error in a test.

data Add tests as long as they induce different program state. For example, add a test if the program internally computes a variable value or a data structure that no previous test input induced.

coverage Assume that if 2 tests yield the same coverage, they are redundant with one another. "Structural coverage" means code coverage. By far the most commonly used one is statement coverage: the set of lines executed by the test input. Sometimes programmers use branch coverage: the set of branches (including their direction, toward then or else) executed by the test input. Software engineering researchers consider other types, such as condition coverage (the values produced by all boolean (sub)expressions in conditional statements) and path coverage (the set of paths executed by the test suite), but these are not of practical interest to programmers.

The "partition the input" and "data" heuristic just above can be viewed as coverage, but for inputs and for variable values. Any dynamic analysis of a program can be used as the basis of a different type of coverage.

co-occurrence of failures If, in the past, two tests always succeeded together or always failed together, then it is reasonable to view one as redundant and discard it, or run it less frequently.

input size A way to achieve a kind of exhaustive testing is to test on every input up to a given size. The "small scope hypothesis" posits that most defects are revealed by small inputs [8], which makes this approach attractive.

"Model checking" is a fancy technical term for "brute-force exploration of all inputs" (up to some bounded size or some other limits, when the space of all inputs is infinite). The research challenge of model checking is to perform the exploration efficiently. This is a deep and challenging research topic.

3.5 Test suite quality (test effectiveness): how good is your test suite?

Programmers want to measure the quality of their test suites. This permits them to compare two test suites or to know when to stop writing tests.

The only thing that matters about a test suite is whether it exposes real defects (that users would observe) as test failures. However, a passing test suite exposes no test failures (programmers fix bugs whenever their test suite exposes them), and the full set of undiscovered defects in the program is unknown, as are the mistakes programmers will make in the program in the future. Therefore, some other metric is needed to compare the sorts of tests that exist in practice.

Any of the above heuristics could be used. In practice, the only widely used test quality heuristic is coverage. Coverage is intuitively attractive: a test suite that executes statement *s* is better than one that does not execute *s* but is otherwise identical, because the former has a chance of revealing a defect in *s*, but the latter does not. However, the correlation is far from perfect. Coverage is used because it is very easy to measure.

Researchers also wish to compare test suite quality, for example to claim that one test generation tool is better than another. One credible approach is to run the tool on old versions of the software to see whether it detects errors that the maintainers later fixed. Another approach, called mutation analysis, is to measure how many *fake* defects a test suite can detect. The researcher creates many variants of the program, each with small change such as changing + to - or changing < to <=. Each variant is called a "mutant", and the "mutation score" tells how many of the fake bugs the test suite detected.

The most important thing to remember about these metrics is that they are all proxies or approximations to the only thing that programmers care about, which is preventing users from observing failures. (No one has ever cared about test coverage itself; they use coverage because they believe it is an (imperfect) proxy for whether the test suite will detect the sorts of real defects that programmers commit.) So, take these measures with a grain of salt.

3.6 Test suite size vs. coverage vs. effectiveness

These are notes on the paper "Revisiting the Relationship Between Fault Detection, Test Adequacy Criteria, and Test Set Size" [2] by Chen et al., in ASE 2020.

This is a somewhat difficult paper to read, especially sections 2.1–2.4 and sections 5.1–5.3 — some readers will want to skip those sections. Here is background information that will help you when you read the paper.

Programmers wish to have good test suites. A good test suite is one that detects real defects that users would observe as failures, and that does so at moderate cost. Detection of defects is called "effectiveness" of a test suite. The real defects in a program are unknowable, so programmers use proxy measures, such as coverage. (The paper uses the term "adequacy" instead of "coverage", in order to encompass mutation score as well as structural coverage; this section uses the more familiar "coverage" where the paper uses "adequacy".) There are many types of coverage, such as statement, branch, and mutation coverage. Programmers measure coverage, believing that greater coverage implies a greater chance of detecting real defects. There is an intuitive argument that this is true: a test can only detect defects on lines it executes, so executing more lines gives a test suite the chance to detect more defects. Given a test suite, adding an additional test that increases coverage makes the suite detect more defects, but it also makes the suite more expensive. To control cost, sometimes a *subsumed* test will be removed after the new test is added.

Experiments show that, on average, a test suite with 100% mutation coverage detects more defects than one with 100% statement coverage. Experiments also show that a test suite with 90% statement coverage detects more defects than one with 80% statement coverage. But many such experiments measure only the *benefits* of the test suite — the defects detected — but not the *costs* of the test suite — the time to run it (and also the effort to construct it). In these experiments, the better test suites turn out to be larger (they contain more tests and run longer) than the worse test suites. Maybe the benefits that were measured were actually due to size rather than to coverage, and the previous experiments overstated the benefit of coverage.

To address this concern, researchers began to do experiments that control for size. The purpose of the Chen et al. paper is to criticize those experiments. The typical setup of those experiments (described as "RANDOMSELECTION" in section 3 of the paper) is to start with an enormous pool of pre-generated tests. The experiment creates test suites by randomly selecting from the pool. Then, the experiment compares the size, coverage, and effectiveness of these suites.

Section 4 points out two problems with the way the suites were generated. One problem (section 4.1) is that many experiments create random suites of a specific size, in order to compare against coverage-complete suites of the same size. (That is: create a coverage-complete ("adequate") test suite, then create a random suite of the same size.) But no practitioner uses test suite size to decide when to stop testing. The second problem (section 4.2) is that no one uses purely random selection to create a test suite.

Section 5 criticizes statistical techniques used in previous studies. Those with statistical background should read it; those without statistical background are best off reading only section 5.0, and skipping sections 5.1–5.3.

Section 6 proposes gives two different approach for controlling for test suite size. The first is to compare suites of the same size. Instead of only considering the final, adequate test suite, consider all the incomplete (inadequate) test suites that were created along the way. The second is to measure probabilistic coupling. Given a defect f and an adequacy goal (such as coverage), probabilistic coupling measures the likelihood that a random test that satisfies the test detects the defect (that is, the test observes a failure resulting from the defect).

One beef I have with this paper is its use of test suite size as a metric. The paper admits that no one cares about this (people care about test execution time), but uses the metric for consistency with previous work. A better metric is execution time. Different tests in a suite often have very different execution time. Just as larger suites tend to have higher coverage, larger (longer-running) tests also have higher coverage. The paper should have used both measurements, to encourage future work to use the correct one.

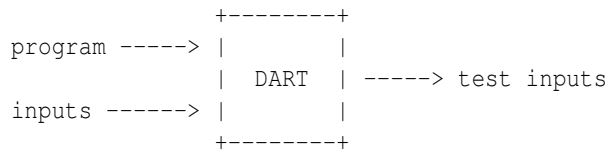
A test selection strategy is a way of reducing a test suite — of selecting a subset of its tests (typically to reduce test execution costs). A test prioritization strategy orders tests with the goal of causing failures to occur as early as possible. Every test prioritization strategy is a test selection strategy: select the highest-ranked tests until the test budget is exceeded. Many techniques that report good results (such as "choose the test that improves coverage the most") are equivalent to "choose the biggest or longest-running test". If they had used execution time rather than number of tests, they would have noticed this fallacy and produced more informative research.

3.7 DART, whitebox fuzzing, and concolic execution

This is a discussion of the paper "DART: Directed automated random testing" [7]. If, after reading the paper, you can explain the relevance of each of the four words in its title, then you understand the key ideas of the paper.

The goal of DART is to create test inputs, with the aim of achieving complete branch coverage. (That is, every boolean conditional in the program evaluates to true at least once during execution of the test suite, and it evaluates to

false at least once.) DART uses implicit oracles: it reports program crashes, which can be due to segmentation faults or to assertions in the program that fail. DART starts from at least one input, and creates more inputs:



3.7.1 Terminology

A *path* is a record of a program execution: a sequence of statements in the program that were executed one after another. A test suite achieves *path coverage* if it executes every path in the program at least once. For the purposes of path coverage, the paths are intraprocedural and do not enter or exit loops; this makes the set of paths finite.

A *path condition* is the sequence of if tests (conditional expressions), and their outcomes, that causes a given path to be taken. For example, consider the following program:

```
if (x < 0)
  print "bad input"
else if (x > 0)
  print "success"
else
  print "failure"
```

There are three paths through the program, and their path conditions are

- x < 0 = true
- x < 0 = false $\wedge$ x > 0 = true
- x < 0 = false $\wedge$ x > 0 = false

3.7.2 An algorithm for path coverage

To understand how DART operates, first consider this naive algorithm to generate a test suite that achieves path coverage.

For every possible path *p*:

1. Determine its path condition.
2. Solve that path condition with a solver, yielding an input that executes the path.

   For example, given the input "x < 0 = false $\wedge$ x > 0 = true", a solver might output "x = 73".

This algorithm yields a set of inputs that, collectively, executes every path.
What could go wrong with this algorithm?

- There may be infeasible paths. Consider the following program:

  ```
  if debug
    print "some diagnostic"
  ...
  if debug
    print "another diagnostic"
  ```

  There are four paths through this program, but two of them are *infeasible* (e.g., debug = true $\wedge$ debug = false); that is, there is no real execution that induces that path. In this trivial example with identical conditional expressions, it is easy to see that the path condition is infeasible, but real examples can be much more subtle.

- The solver may not be powerful enough to solve the path condition.
- There may be so many paths that the algorithm has no hope of processing all of them in a reasonable amount of time.

3.7.3 The DART algorithm

DART's key idea is to do a hybrid concrete–symbolic analysis: where the solver is unable to produce a solution, DART uses data from a concrete execution instead. More specifically, given a concrete execution, DART tries to use a solver to create another concrete execution that is similar, but not identical, to it.

Here is the DART algorithm. It repeats the following for as long as time allows.

1. Run the program once.
   The input could be (for example) a random input, an existing test input, or an input obtained from in-field profiling.
2. Let C be the path condition for the specific path that was executed.
   Let V be the set of variables used in C, and for each variable let $concrete(v)$ be the value the variable took on at run time.
3. For each condition (boolean expression) in the path condition C, try to find a new test input that takes the branch in the opposite direction.

   for i = 0 to $|C| - 1$
   # Create a new path condition that negates $C[i]$ and discards everything after $C[i]$.
   $C' = C[0..i-1] + \neg C[i]$
   # Try to solve the new path condition, obtaining values for as many variables as possible.
   $solvedValues = partialSolve(C')$
   # If some parts of the new path condition are not solvable, use values from the concrete execution.
   $varValues = solvedValues + \{v \mapsto concrete(v) : v \in V - keys(solvedValues)\}$

   The advantage of this approach is that it overcomes a key limitation of solvers: they fail for many real-world programs, for instance when those programs use pointers/references.

   A disadvantage of this approach is that the this simplified condition may be unsolvable/infeasible, even if the original condition was not.

As an example, consider the following program:

```
if x > 0
  if y == x + z
    if f(z) is even
      g(x, y, z)
    else
      h(z)
```

Suppose that there is a concrete execution that executes $\mathsf{h(x)}$. DART would try to negate the last condition in the path, leading to this path condition:

$$\mathsf{x} > 0 \wedge \mathsf{y} = \mathsf{x} + \mathsf{z} \wedge \mathsf{f(z)} \text{ is even} .$$

Suppose the solver cannot solve "$\mathsf{f(z)}$ is even", and perhaps other parts of the path condition. Then, by substituting in concrete values observed during the concrete execution, the effective path condition may become

$$\mathsf{x} > 0 \wedge \mathbf{5} = \mathsf{x} + \mathbf{12} \wedge \mathsf{f}(\mathbf{12}) \text{ is even} .$$

Determining whether $\mathsf{f}(12)$ is even is easy to do (though if it is not, then the path condition is insoluble).

In practice, a solver tool reports a solution or reports "I don't know". A tool generally cannot know that a problem is insoluble, only that the tool can't solve it.

Some later papers by the same authors use the terms "whitebox fuzzing" [6] and "concolic testing" [12], which are synonyms. "Concolic" is a portmanteau of "concrete" and "symbolic". "Concolic testing" is a misnomer; the term should really be "concolic execution", and testing is one application of it.

Exercise 31 *DART executes a program both concretely and symbolically. Briefly explain the difference between these two types of executions in DART and why both are necessary.*

Exercise 32 *Give one example for path constraints that DART can solve and one example for path constraints that it cannot. Briefly explain why.*

Exercise 33 *Give an example of a test oracle used by test cases generated by DART. Give one example of a test oracle that DART cannot automatically generate and briefly explain why.*

Exercise 34 *When DART generates a new test input, it is similar to an existing execution. What are the advantages and disadvantages of that similarity?*

Appendix A

Terminology

A.1 Precision and recall

Given a tool or technique in program analysis, how good are its results? In order to compare different tools, we need to quantify the quality of their results using precision and recall. Precision indicates how much of the tool output is correct: $\frac{\text{correct tool outputs}}{\text{all tool outputs}}$. Recall indicates how much of the goal output is in the tool output: $\frac{\text{correct tool outputs}}{\text{goal outputs}}$. See below for formal definitions.

A perfect tool has 100% precision and 100% recall. Unfortunately, it is a theorem (following from the halting problem) that no program analysis tool has 100% precision and 100% recall on all programs.

Precision and recall are the appropriate measures for an information retrieval problem [11, 13]. In an information retrieval problem, the tool's output is a set of values. The ground truth is also a set of values. Precision and recall are two complementary ways to compare these sets.

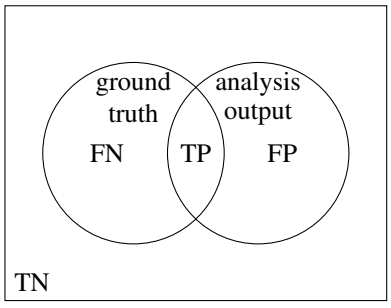
Another way to represent a set is an indicator function. An indicator function maps every value that could possibly be in the set to either one or zero, indicating whether the value is in the set or not. In program analysis, an indicator function's boolean output is labeled as "positive" and "negative" rather than "1" and "0", since 1 and 0 can be ambiguous.

The ground truth labels each possible value as positive or negative, and the tool's output also labels each possible value as positive or negative.

A true positive or TP is when the tool correctly labels a value as positive.
A false positive or FP is when the tool incorrectly labels a value as positive.
A true negative or TN is when the tool correctly labels a value as negative.
A false negative or FN is when the tool incorrectly labels a value as negative.

Here are two ways to view the 4 possibilities:

		analysis output: positive	analysis output: negative
ground truth	positive	TP	FN
ground truth	negative	FP	TN



Now, we can define precision and recall.

$$\text{precision} = \frac{|\text{TP}||}{|\text{TP}| + |\text{FP}|}$$

$$\text{recall} = \frac{|\text{TP}|}{|\text{TP}| + |\text{FN}|}$$

Example. There are 151 Pokémon in the first generation. Suppose that a tool reports catching 42 first-generation Pokémon, but the tool's output includes Snubbull and Granbull, which are second-generation Pokémon. Then:

$$\begin{aligned} |\text{TP}| &= 40 \\ |\text{FP}| &= 2 \\ |\text{TN}| &= 111 \\ \text{precision} &= \frac{|\text{TP}|}{|\text{TP}|+|\text{FP}|} = \frac{40}{40+2} = \frac{40}{42} = 95\% \\ \text{recall} &= \frac{|\text{TP}|}{|\text{TP}|+|\text{FN}|} = \frac{40}{40+111} = \frac{40}{151} = 26\% \end{aligned}$$

A.1.1 Other terms related to precision and recall

For program analysis tools that aim to detect program defects, an output is a warning about a program defect. A false positive is usually called a "false alarm", and a false negative is usually called a "missed alarm".

The F-score, F1-score, or F-measure is a single number that weights precision and recall equally. It is defined as the harmonic mean of precision and recall:

$$F_1 = \frac{2 \cdot \text{TP}}{2 \cdot \text{TP} + \text{FP} + \text{FN}}$$

It is rarely the case that users are equally concerned about false positives and false negatives, so it is rarely the case that the F-score is the best way to evaluate a program analysis tool. Sometimes people use the term "accuracy" as a synonym for F-score, but this is a misuse of "accuracy". "Accuracy" is so ambiguous (used with many different meanings) that it is best to avoid the term.

Here are the meanings of "precision" and "accuracy" in normal English usage. These are related to, but not identical to, their use in program analysis. Accuracy measures how close results are to the true value; it is a measure of correctness. Precision measures how close results are to one another; it is a measure of reproducibility. Consider this example:

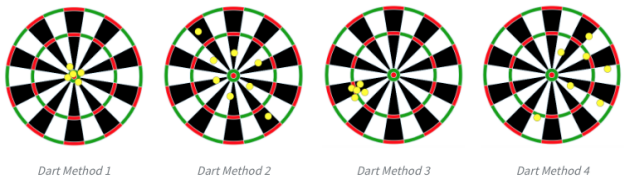


Image credit: By https://commons.wikimedia.org/wiki/User:Arbeck - File:Accuracy_and_Precision.svg, CC BY 4.0, https://commons.wikimedia.org/w/index.php?curid=133410581.

Method 1 is accurate and precise.
Method 2 is accurate but not precise.
Method 3 is precise but not accurate.
Method 4 is neither precise nor accurate.

A "type I error" is a false positive, and a "type II error" is a false negative. Unfortunately, these terms convey no intuition, so it is best to avoid them.

A.2 Soundness and completeness

Consider a program analysis tool that aims to detect program defects. Equivalently, its goal is to verify a program as correct (with respect to some specific program property, such as null pointer exceptions or out-of-bounds indexes).

A *sound* tool is one that detects every defect. A sound tool has no false negatives, so by definition its recall is 100%. If a sound tool issues no warnings, then the program is correct (with respect to the program properties assessed by the

analysis tool). A sound tool may issue false positive warnings. A conservative analysis is the typical way to implement a sound tool.

A *complete* tool is one that never mis-reports a warning. It suffers no false positives, so by definition its precision is 100%. A complete tool may suffer false negatives (missed alarms). Complete tools are rare.

Beware: Different communities use the terms soundness and completeness to mean exactly the opposite concepts! In particular, depending on whether the goal is to prove correctness or to find defects, the community may use "sound" and "complete" in different ways. In program analysis, it is usually best to use soundness as defined above (and to clarify that definition in your writing), and not to use the term completeness.

A.3 Bugs, defects, and failures

The term "bug" is ambiguously used with multiple different meanings. For clarity, it is best to avoid that word, except in the multiword expression "bug report". The word "fault" is also ambiguous. Here are better, standard terms to use:

defect A defect is a flaw, failing, or imperfection in a system, such as an incorrect design, algorithm, or implementation. It is typically caused by a human mistake. It is also known as a fault, and this is the most common meaning for the term "bug", but those terms are ambiguous and should be avoided.

error An error is a discrepancy between the intended behavior of a system and its actual behavior *within* the system, such as an incorrect program state (= incorrect variable value) or an invocation of a procedure that is contrary to the procedure's specification or contract. Many errors are not detectable without observing internal state, such as via an assert statement or a debugger.

failure A failure is an instance in time when a system displays behavior that is contrary to its specification. It is a user-visible error. A non-user-visible error frequently leads to a failure later in the execution.

Appendix B

Solutions to selected exercises

1 A larger lattice enables a more precise analysis, but that more precise analysis is more expensive to compute.

3 $\mathsf{y} = y_1$, and x is twice that. (x is also even, which is implied by the fact that $\mathsf{x} = \mathsf{y} \cdot 2$.)

If you didn't get this as your answer, redo the problem until you do.

Here are two ways to obtain that solution via "symbolic execution". One way is to work forward from the beginning, using symbolic expressions for each variable. Represent the current state of the program as a map from variables to values.

```
x1 = 0;
                          [ x1 -> 0, y1 -> ? ]
y1 = read_even_integer();
                          [ x1 -> 0, y1 -> ? ]
x2 = y1+1;
                          [ x2 -> y1 + 1, y1 -> ? ]
y2 = 2*x2;
                          [ x2 -> y1 + 1, y2 -> 2 * y1 + 2 ]
x3 = y2-2;
                          [ x3 -> 2 * y1, y2 -> 2 * y1 + 2 ]
y3 = x3/2;
                          [ x3 -> 2 * y1, y3 -> y1 ]
```

The other approach is to work backward from the final expressions for x3 and y3:

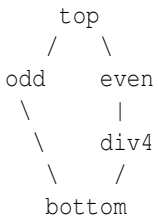
```
y3 = x3 / 2
y3 = (y2 - 2) / 2
y3 = (2 * x2 - 2) / 2
y3 = (2 * (y1 + 1) - 2) / 2
y3 = (2 * y1 + 2 - 2) / 2
y3 = y1
x3 = y1 * 2
```

4 x is even, and y is unknown ($\top$).

Note that the fact that y is even is expressible in this analysis, but the analysis cannot discover this true fact! The analysis loses information and its final estimate for y is "unknown" instead of "even".

5 The lattice points are $\top$, "divisible by 4", "even", "odd", and $\bot$.

The lattice is:



6 The abstract store at the end of the loop is $[\mathsf{x} \to \mathsf{even}, \mathsf{y} \to \mathsf{unknown}, \mathsf{result} \to \mathsf{unknown}]$.

7 The abstract store at the end of the loop is

1. $[\mathsf{x} \to \mathsf{even}, \mathsf{y} \to \mathsf{even}, \mathsf{result} \to \mathsf{even}]$
2. $[\mathsf{x} \to \mathsf{even}, \mathsf{y} \to \mathsf{even}, \mathsf{result} \to \mathsf{even}]$
3. $[\mathsf{x} \to \mathsf{even}, \mathsf{y} \to \mathsf{odd}, \mathsf{result} \to \mathsf{unknown}]$
4. $[\mathsf{x} \to \mathsf{even}, \mathsf{y} \to \mathsf{odd}, \mathsf{result} \to \mathsf{unknown}]$

The last two results are imprecise, because $\mathsf{even} \times \mathsf{odd} = \mathsf{even}$ and $\mathsf{odd} \times \mathsf{odd} = \mathsf{odd}$. Can you think of an analysis that could produce a more precise results?

9 The analysis does not terminate: the symbolic expressions grow without bound.

11 On every iteration through the loop, some value in the abstract store changes. (Otherwise, the fixpoint iteration would stop.) Every change is in the upward direction. In the worst case, the initial abstract store maps every variable to $\bot$, every iteration through the loop changes only one variable's abstract value by moving it upward just one point in the lattice, and the final abstract store maps every variable to $\top$.

12 The lattice is:

In addition to the points mentioned in the exercise, this lattice also contains $\top$ and $\bot$. Every lattice must have a unique top element (to represent "unknown" or "no information is known") and a unique bottom element (to represent "dead code" or "the analysis has not yet reached this value").

13

+	$\top$	odd	even	mod4	is2	$\bot$
$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\bot$
odd	$\top$	even	odd	odd	odd	$\bot$
even	$\top$	odd	even	even	even	$\bot$
mod4	$\top$	odd	even	mod4	even	$\bot$
is2	$\top$	odd	even	even	mod4	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

Think about why $+(\top, \bot) = \bot$ is a better choice than $+(\top, \bot) = \top$. The latter choice is not incorrect, but it is imprecise.

14 Our solution uses the following template. Since g has only one variable, the abstract store is a singleton mapping. Each statement is numbered.

```
def g() {
                    [ x -> ____ ]
  x = 2;                            (1)
                    [ x -> ____ ]
  while                             (2)
                    [ x -> ____ ]     // this is a merge point (2 predecessors), and it has 2 successors
        (x < 10) {                  (3)
                    [ x -> ____ ]
    x = x + 2;                      (4)
                    [ x -> ____ ]
  }
                    [ x -> ____ ]
  return x;                         (5)
                    [ x -> ____ ]
}

Worklist: ____________________________________________
```

The successors of (3) — that is, the successors of the while condition — are statements (4) and (5). The successor of statement (4) is (3). Statement (3) has two predecessors, (2) and (4).

Initially, all abstract stores are bottom; that is, they map all values to bottom. (If there were formal parameters, they would map to top, not bottom, in the store at the beginning of the procedure body.) The initial worklist is the entry statement. (It also works to make the initial worklist all statements.)

```
def g() {
                    [ x -> bottom ]
  x = 2;                            (1)
                    [ x -> bottom ]
  while                             (2)
                    [ x -> bottom ]     // this is a merge point (2 predecessors), and it has 2 successors
        (x < 10) {                  (3)
                    [ x -> bottom ]
    x = x + 2;                      (4)
                    [ x -> bottom ]
  }
                    [ x -> bottom ]
  return x;                         (5)
                    [ x -> bottom ]
}

Worklist: (1)
```

The abstract interpretation selects a statement from the worklist, analyses it, possibly puts new program points on the worklist, and continues until the worklist is empty.

In the following, differences from the previous analysis are shown CAPITALIZED.

Choose statement (1) from the worklist. Apply its transfer function, and update the following store(s).

Since the post-store at statement (1) changed, add all of its successors — namely, statement (2) — to the worklist.

```
def g() {
                    [ x -> bottom ]
  x = 2;                            (1)
                    [ x -> IS2    ]
  while                             (2)
```

```
                    [ x -> bottom ]
        (x < 10) {                  (3)
                    [ x -> bottom ]
    x = x + 2;                      (4)
                    [ x -> bottom ]
  }
                    [ x -> bottom ]
  return x;                         (5)
                    [ x -> bottom ]
}

Worklist: (2)
```

Choose statement (2) from the worklist. Its post-state feeds into a merge point. (This would be clearer if we had used if and goto statements rather than the while construct.) To obtain a pre-store for statement (3), lub the post-stores for statements (2) and (4). Those stores are $[x \to is2]$ and $[x \to \bot]$, respectively, and their lub is $[x \to is2]$.

Since the post-store of statement (2) changed, add its successors to the worklist.

```
def g() {
                    [ x -> bottom ]
  x = 2;                            (1)
                    [ x -> is2    ]
  while                             (2)
                    [ x -> IS2    ]
        (x < 10) {                  (3)
                    [ x -> bottom ]
    x = x + 2;                      (4)
                    [ x -> bottom ]
  }
                    [ x -> bottom ]
  return x;                         (5)
                    [ x -> bottom ]
}

Worklist: (3)
```

It may seem alarming that the algorithm's current estimate for x in x < 10 is that the value of x is exactly 2. The algorithm doesn't report any results until it has reached a fixed point, at which time its results are sound.

Choose statement (3) from the worklist. This predicate enforces no information about x, so propagate the pre-store to before all the successors of statement (3), and add those successors to the worklist.

```
def g() {
                    [ x -> bottom ]
  x = 2;                            (1)
                    [ x -> is2    ]
  while                             (2)
                    [ x -> is2    ]
        (x < 10) {                  (3)
                    [ x -> IS2    ]
    x = x + 2;                      (4)
                    [ x -> bottom ]
  }
                    [ x -> IS2    ]
```

```
  return x;                          (5)
                    [ x -> bottom ]
}

Worklist: (4), (5)
```

Choose statement (4) from the worklist. Apply its transfer function:

```
def g() {
                    [ x -> bottom ]
  x = 2;                             (1)
                    [ x -> is2    ]
  while                              (2)
                    [ x -> is2    ]
        (x < 10) {                   (3)
                    [ x -> is2    ]
    x = x + 2;                       (4)
                    [ x -> MOD4   ]
  }
                    [ x -> is2    ]
  return x;                          (5)
                    [ x -> bottom ]
}

Worklist: (5)
```

and propagate that to before its successor. Statement (3) has two predecessors, so its pre-state is the lub of the post-state of statements (1) and (4). Those post-states are, respectively, $[x \rightarrow is2]$ and $[x \rightarrow mod4]$, and their lub is $[x \rightarrow even]$. Since that value has changed, add the successor, statement (3), to the worklist.

```
def g() {
                    [ x -> bottom ]
  x = 2;                             (1)
                    [ x -> is2    ]
  while                              (2)
                    [ x -> EVEN   ]
        (x < 10) {                   (3)
                    [ x -> is2    ]
    x = x + 2;                       (4)
                    [ x -> mod4   ]
  }
                    [ x -> is2    ]
  return x;                          (5)
                    [ x -> bottom ]
}

Worklist: (5), (3)
```

Choose statement (3) from the worklist. (If you choose statement (5), the fixed point is still reached, but it takes longer.)

Statement (3) has no effect on the abstract store, so propagate the store to the two successors, and add the successors to the worklist. Note that no lub is performed, because no merge point is encountered.

```
def g() {
                    [ x -> bottom ]
  x = 2;                            (1)
                    [ x -> is2    ]
  while                             (2)
                    [ x -> even   ]
        (x < 10) {                  (3)
                    [ x -> EVEN   ]
    x = x + 2;                      (4)
                    [ x -> mod4   ]
  }
                    [ x -> EVEN   ]
  return x;                         (5)
                    [ x -> bottom ]
}

Worklist: (5), (4)
```

Choose statement (4) from the worklist. Apply its transfer function. Its poststate is unchanged, so do not add statement (4)'s successors to the worklist.

```
def g() {
                    [ x -> bottom ]
  x = 2;                            (1)
                    [ x -> is2    ]
  while                             (2)
                    [ x -> even   ]
        (x < 10) {                  (3)
                    [ x -> even   ]
    x = x + 2;                      (4)
                    [ x -> mod4   ]     // no change here!
  }
                    [ x -> even   ]
  return x;                         (5)
                    [ x -> bottom ]
}

Worklist: (5)
```

Choose statement (5) from the worklist. It does not affect the store; propagate the store to its poststate. It has no successors, so add nothing to the worklist. The worklist is empty and the algorithm has completed. The algorithm has computed a pre- and post-store for every statement.

```
def g() {
                    [ x -> bottom ]
  x = 2;                            (1)
                    [ x -> is2    ]
  while                             (2)
                    [ x -> even   ]
        (x < 10) {                  (3)
                    [ x -> even   ]
    x = x + 2;                      (4)
                    [ x -> mod4   ]     // no change here!
```

```
  }
                    [ x -> even   ]
  return x;                        (5)
                    [ x -> EVEN   ]
}

Worklist:
```

17 The lub function must be monotonic to guarantee that the analysis terminates (given a finite-height lattice). As an example, consider the following lattice. It is nonsensical because lub is not monotonic.

$$\begin{aligned} \text{Lattice points} &= \{\top, \bot\} \\ \\ \text{lub}(\top, \top) &= \bot \\ \text{lub}(\top, \bot) &= \top \\ \text{lub}(\bot, \top) &= \top \\ \text{lub}(\bot, \bot) &= \top \end{aligned}$$

Also consider this code to analyze:

```
x = input()
loop:
goto loop
```

which can be equivalently expressed as:

```
x = input()
label:
if (some-unanalyzable-expression)
  goto label
```

The CFG looks like this (the letter "v" is an arrowhead):

```
 _   |
/ \  |
|  v v
|  join
|   |
|   v
|  nop
\_/  |
     v
```

The estimate for x starts out as $\top$ after the input statement, but on every iteration through the loop it flip-flops and the analysis never terminates.

21 The **lattice** is

```
top = unknown = the variable's value may or may not have been used so far
 |
used = the variable's value has definitely used
```

A more precise analysis (or one intended to answer different questions) might include a "definitely not used" lattice element, but we don't need one to answer our questions.

The **transfer function** for `x = y + z` affects the abstract value of `x`, `y`, and `z`. (Here, `x`, `y`, and `z` are called meta-variables, since you can substitute in any variable for them.)

$$\text{store}_{post} = \text{store}_{pre}[\mathsf{y} := \text{used}][\mathsf{z} := \text{used}][\mathsf{x} := \text{top}]$$

The notation $map[k := v]$ means a map just like *map*, but with k mapped to v.

The rationale for the rule is that after the statement `x = y + z`, the values of `y` and `z` have definitely been used, but the value of `x` has not yet been used. Note that the transfer function doesn't depend on the previous abstract values of the variables in the expression.

Note that the similar rule

$$\text{store}_{post} = \text{store}_{pre}[\mathsf{x} := \text{top}][\mathsf{y} := \text{used}][\mathsf{z} := \text{used}]$$

would be incorrect, because when applied to `i = i + 1` the store would map `i` to used.

The **transfer function** for `return x` is

$$\text{store}_{post} = \text{store}_{pre}[\mathsf{x} := \text{used}]$$

The **abstract stores** produced by analyzing the function are:

```
def f(a, b, c) {
                [ a -> top , b -> top , c -> top , d -> top  ] // nothing has been used yet
  if (*) {
                [ a -> top , b -> top , c -> top , d -> top  ] // same as before if statement
    d = a + b;
                [ a -> used, b -> used, c -> top , d -> top  ] // apply xfer fn for "x=y+z"
  } else {
                [ a -> top , b -> top , c -> top , d -> top  ] // same as before if statement
    d = a + c;
                [ a -> used, b -> top , c -> used, d -> top  ] // apply xfer fn for "x=y+z"
  }
                [ a -> used, b -> top , c -> top , d -> top  ] // merge (lub) of stores at end of branches
  return d;
                [ a -> used, b -> top , c -> top , d -> used ] // apply xfer fn for "return x"
}
```

The most interesting program point is the end of the `if` statement, which is a merge point for control from the two branches. Therefore, the abstract store there is the lub of the abstract stores at the end of each of the branches. To lub two abstract stores, lub their values pointwise. The abstract store at the end of the `if` statement is

$$[\mathsf{a} \to \text{used}, \mathsf{b} \to \text{top}, \mathsf{c} \to \text{top}, \mathsf{d} \to \text{top}]$$

which indicates that at that point, the current value of variable `a` has definitely been used, and no other variable's value has definitely been used. Neither `b` nor `c` has been definitely used: for each of those variables, there is some path on which it is not used.

22 Use glb instead of lub. The rest of the abstract interpretation is unchanged.

Applying glb to the then and else states at the join point:

$$\begin{aligned} \text{then} &: [\mathsf{a} \to \text{used}, \mathsf{b} \to \text{used}, \mathsf{c} \to \top, \mathsf{d} \to \top] \\ \text{else} &: [\mathsf{a} \to \text{used}, \mathsf{b} \to \top, \mathsf{c} \to \text{used}, \mathsf{d} \to \top] \\ \text{glb} &: [\mathsf{a} \to \text{used}, \mathsf{b} \to \text{used}, \mathsf{c} \to \text{used}, \mathsf{d} \to \top] \end{aligned}$$

The final answer at the end of the procedure is

$$[\mathsf{a} \to \text{used}, \mathsf{b} \to \text{used}, \mathsf{c} \to \text{used}, \mathsf{d} \to \text{used}]$$

In other words, `a`, `b`, `c`, and `d` might be used by the end of the procedure.

23 A sound analysis reports either a property that exactly reflects all possible run-time behaviors, or an over-approximation of run-time behaviors. It is always sound (but not precise) to report a higher point in the lattice than the least fixed point.

34 Advantage: The new test input is likely to be realistic, if the original input was realistic.

Advantage: The new test input is more likely to be feasible (compared to a test input generated by solving the path conditions of an arbitrary path), since all but the last condition is definitely feasible.

Advantage: The new test input can be used for fault localization: there are two inputs that execute similar paths (though the inputs might not be otherwise similar).

Disadvantage: The new test input provides little new coverage and executes little new functionality.

Appendix C

Acknowledgments

Thanks to René Just, Martin Kellogg, and Ardi Madadi for comments and corrections. Any remaining errors are the fault of the author.

Bibliography

[1] Henry Baker. Unify and conquer (garbage, updating, aliasing, ...). In *LFP '90: Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pages 218–226, Nice, France, June 1990.

[2] Yiqun T. Chen, Rahul Gopinath, Anita Tadakamalla, Michael D. Ernst, Reid Holmes, Gordon Fraser, Paul Ammann, and René Just. Revisiting the relationship between fault detection, test adequacy criteria, and test set size. In *ASE 2020: Proceedings of the 35th Annual International Conference on Automated Software Engineering*, pages 237–249, Melbourne, Australia, September 2020.

[3] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, CA, January 1977.

[4] Edsger W. Dijkstra. Ewd303: On the reliability of programs. n.d.

[5] Michael D. Ernst. Static and dynamic analysis: Synergy and duality. In *WODA 2003: Workshop on Dynamic Analysis*, pages 24–27, Portland, OR, USA, May 2003.

[6] Patrice Godefroid, Adam Kieżun, and Michael Y. Levin. Grammar-based whitebox fuzzing. In *PLDI 2008: Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation*, pages 206–215, Tucson, AZ, USA, June 2008.

[7] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *PLDI 2005: Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 213–223, Chicago, IL, USA, June 2005.

[8] Daniel Jackson and Craig A. Damon. Elements of style: Analyzing a software design feature with a counterexample detector. *IEEE Transactions on Software Engineering*, 22(7):484–495, 1996.

[9] Robert O'Callahan and Daniel Jackson. Lackwit: A program understanding tool based on type inference. In *ICSE '97: Proceedings of the 19th International Conference on Software Engineering*, pages 338–348, Boston, MA, May 1997.

[10] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins, and Michael D. Ernst. Practical pluggable types for Java. In *ISSTA 2008, Proceedings of the 2008 International Symposium on Software Testing and Analysis*, pages 201–212, Seattle, WA, USA, July 2008.

[11] Gerard Salton. *Automatic Information Organization and Retrieval*. McGraw-Hill, 1968.

[12] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. In *ESEC/FSE 2005: Proceedings of the 10th European Software Engineering Conference and the 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 263–272, Lisbon, Portugal, September 2005.

[13] Cornelis Joost van Rijsbergen. *Information Retrieval*. Butterworths, London, second edition, 1979.

Index