Static Program Analysis

Anders Møller and Michael I. Schwartzbach

January 15, 2018

Department of Computer Science
Aarhus University, Denmark
{amoeller,mis}@cs.au.dk

Contents

Preface

Static program analysis is the art of reasoning about the behavior of computer programs without actually running them. This is useful not only in optimizing compilers for producing efficient code but also for automatic error detection and other tools that can help programmers. A static program analyzer is a program that reasons about the behavior of other programs. For anyone interested in programming, what can be more fun than writing programs that analyze programs?

As known from Turing and Rice, all nontrivial properties of the behavior of programs written in common programming languages are mathematically undecidable. This means that automated reasoning of software generally must involve approximation. It is also well known that testing, i.e. concretely running programs and inspecting the output, may reveal errors but generally cannot show their absence. In contrast, static program analysis can – with the right kind of approximations – check all possible executions of the programs and provide guarantees about their properties. One of the key challenges when developing such analyses is how to ensure high precision and efficiency to be practically useful.

These notes present principles and applications of static analysis of programs. We cover basic type analysis, lattice theory, control flow graphs, dataflow analysis, fixed-point algorithms, narrowing and widening, path sensitivity, interprocedural analysis and context sensitivity, control flow analysis, and several flavors of pointer analysis. A tiny imperative programming language with pointers and first-class functions is subjected to numerous different static analyses illustrating the techniques that are presented.

We emphasize a *constraint-based approach* to static analysis where suitable constraint systems conceptually divide the analysis task into a front-end that generates constraints from program code and a back-end that solves the constraints to produce the analysis results. This approach enables separating the analysis specification, which determines its precision, from the algorithmic aspects that are important for its performance. In practice when implementing analyses, we often solve the constraints on-the-fly, as they are generated, without

representing them explicitly.

We focus on analyses that are fully *automatic* (i.e., not involving programmer guidance, for example in the form of loop invariants) and *conservative* (sound but incomplete), and we only consider Turing complete languages (like most programming languages used in ordinary software development).

The analyses that we cover are expressed using different kinds of constraint systems, each with their own constraint solvers:

- term unification constraints, with an almost-linear union-find algorithm,
- conditional subset constraints, with a cubic algorithm, and
- monotone constraints over lattices, with variations of fixed-point solvers.

The style of presentation is intended to be precise but not overly formal. The readers are assumed to be familiar with advanced programming language concepts and the basics of compiler construction and computability theory.

The notes are accompanied by a web site that provides lecture slides, an implementation (in Scala) of most of the algorithms we cover, and additional exercises:

`http://cs.au.dk/~amoeller/spa/`

Chapter 1

Introduction

Static program analysis has been used since the 1970's in optimizing compilers. More recently, it has proven useful also for bug finding and verification tools and in IDEs to support, for example, navigation, code completion, refactoring, and program understanding. Static program analysis aims to automatically answer questions about a given program. There are many interesting such questions, for example:

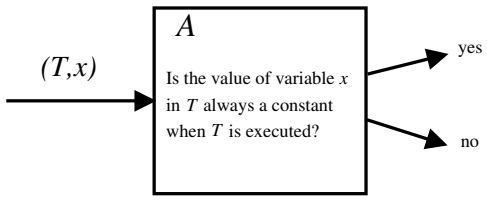
- Can input values from untrusted users flow unchecked to file system operations? (That may have serious security implications.)
- Can secret information become publicly observable? (Similar to the previous example.)
- Does the program terminate on every input? (Most programs are intended to have this property.)
- Can the program deadlock? (This is often a concern for multi-threaded programs that use locks for synchronization.)
- How large can the heap become during execution? (Perhaps the program needs to run on an embedded device where we do not want to put more memory than necessary.)
- Does there exist an input that leads to a null pointer dereference, division-by-zero, or arithmetic overflow? (Such situation are considered errors in most programs, similar to nontermination.)
- Are all assertions guaranteed to succeed? (Assertions express program specific correctness properties that are supposed to hold in all executions.)
- Are all variables initialized before they are read? (If not, the program may be vulnerable to attacks by malicious users.)
- Are arrays always accessed within their bounds? (Similar to the previous example, this is important for security reasons.)

- Can there be dangling references, e.g. pointers to memory that has been freed? (Yet another cause of security issues.)
- Does the program contain dead code, or more specifically, is function `f` unreachable from `main`? (If so, the code size can be reduced.)
- Does the value of variable `x` depend on the program input? (If not, it could be precomputed at compile time.)
- Is it possible that the value of `x` will be read in the future? (If so, it may be worthwhile to cache the value.)
- Do `p` and `q` point to disjoint structures in the heap? (That may enable parallel processing.)
- Are all resources properly released before the program terminates? (Otherwise, the program may run out of resources at some point.)
- What types of values can variable `x` have? (Maybe some of those types are not what the programmer intended.)
- At which program points could `x` be assigned its current value? (Programmers often ask this kind of question when trying to understand large codebases.)
- Which functions may possibly be called on line 117? (Similar to the previous example.)
- What are the lower and upper bounds of the integer variable `x`? (The answer may guide the choice of runtime representation of the variable.)
- Is function `f` always called before function `g`? (Perhaps the documentation of those functions require this to be the case.)
- Can the value of variable `x` affect the value of variable `y`? (Such questions often arise during debugging, when programmers try to understand why a certain bug appears.)

Regarding correctness, programmers routinely use testing to gain confidence that their programs work as intended, but as famously stated by Dijkstra: *"Program testing can be used to show the presence of bugs, but never to show their absence."* Ideally we want guarantees about what our programs may do for all possible inputs, and we want these guarantees to be provided automatically, that is, by programs. A *program analyzer* is such a program that takes other programs as input and decides whether or not they have a given property.

Rice's theorem is a general result from 1953 which informally states that all interesting questions about the behavior of programs (written in Turing-complete programming languages[1]) are *undecidable*. This is easily seen for any special case. Assume for example the existence of an analyzer that decides if a variable in a program has a constant value in any execution. In other words, the analyzer is a program A that takes as input a program T and one of T's variables x, and decides whether or not x has a constant value whenever T is executed.

[1] From this point on, we only consider Turing complete languages.



We could then exploit this analyzer to also decide the halting problem by using as input the following program where `TM(j)` simulates the `j`'th Turing machine on empty input:

```
x = 17; if (TM(j)) x = 18;
```

Here `x` has a constant value if and only if the `j`'th Turing machine does not halt on empty input. If the hypothetical constant-value analyzer A exists, then we have a decision procedure for the halting problem, which is known to be impossible.

This seems like a discouraging result. However, our real goal is not to decide such properties but rather to solve practical problems like making the program run faster or use less space, or finding bugs in the program. The solution is to settle for *approximative* answers that are still precise enough to fuel our applications. While it is impossible to build an analysis that would correctly decide a property for any analyzed program, it is often possible to build analysis tools that give useful answers for most realistic programs. As the ideal analyzer does not exist, there is always room for building more precise approximations (which is colloquially called the *full employment theorem for static program analysis designers*).

Approximative answers may be useful for finding bugs in programs, which may be viewed as a weak form of program verification. As a case in point, consider programming with pointers in the C language. This is fraught with dangers such as `null` dereferences, dangling pointers, leaking memory, and unintended aliases. Ordinary compilers offer little protection from pointer errors. Consider the following small program which may perform every kind of error:

```
int main(int argc, char *argv[]) {
  if (argc == 42) {
    char *p,*q;
    p = NULL;
    printf("%s",p);
    q = (char *)malloc(100);
    p = q;
    free(q);
    *p = 'x';
    free(p);
    p = (char *)malloc(100);
    p = (char *)malloc(100);
    q = p;
```

```
      strcat(p,q);
      assert(argc > 87);
    }
  }
```

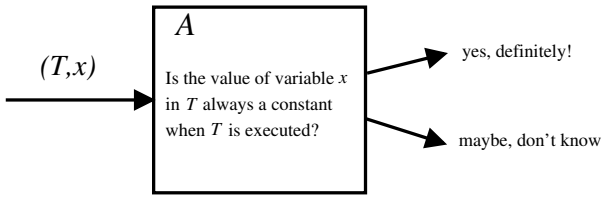
Standard compiler tools such as `gcc -Wall` detect no errors in this program. Finding the errors by testing might miss the errors, unless we happen to have a test case that runs the program with exactly 42 arguments. However, if we had even approximative answers to questions about `null` values, pointer targets, and branch conditions then many of the above errors could be caught statically, without actually running the program.

Exercise 1.1: Describe all the pointer-related errors in the above program.

Ideally, the approximations we use are *conservative* (or *safe*), meaning that all errors lean to the same side, which is determined by our intended application. As an example, approximating the memory usage of programs is conservative if the estimates are never lower than what is actually possible when the programs are executed. Conservative approximations are closely related to the concept of soundness of program analyzers. We say that a program analyzer is *sound* if it never gives incorrect results (but it may answer *maybe*). Thus, the notion of soundness depends on the intended application of the analysis output, which may cause some confusion. For example, a verification tool is typically called sound if it never misses any errors of the kinds it has been designed to detect, but it is allowed to produce spurious warnings (also called false positives), whereas an automated testing tool is called sound if all reported errors are genuine, but it may miss errors.

Program analyses that are used for optimizations typically require soundness. If given false information, the optimization may change the semantics of the program. Conversely, if given trivial information, then the optimization fails to do anything.

Consider again the problem of determining if a variable has a constant value. If our intended application is to perform constant propagation optimization, then the analysis may only answer *yes* if the variable really is a constant and must answer *maybe* if the variable may or may not be a constant. The trivial solution is of course to answer *maybe* all the time, so we are facing the engineering challenge of answering *yes* as often as possible while obtaining a reasonable analysis performance.



In the following chapters we focus on techniques for computing approxima-

tions that are conservative with respect to the semantics of the programming language.

1.1 Undecidability of Program Correctness

(This section requires familiarity with the concept of universal Turing machines; it is not a prerequisite for the following chapters.)

The reduction from the halting problem presented above shows that some static analysis problems are undecidable. However, halting is often the least of the concerns programmers have about whether their programs work correctly. For example, if we wish to ensure that the programs we write cannot crash with null pointer errors, we may be willing to assume that the programs do not also have problems with infinite loops.

Using a diagonalization argument we can show a very strong result: It is impossible to build a static program analysis that can decide whether a given program may fail when executed. Moreover, this result holds even if the analysis is only required to work for programs that halt on all inputs. The halting problem is not the only obstacle. Approximation is inevitable.

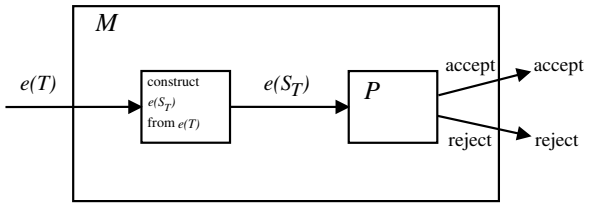
If we model programs as deterministic Turing machines, program failure can be modeled using a special *fail* state.[2] That is, on a given input, a Turing machine will eventually halt in its accept state (intuitively returning "yes"), in its reject state (intuitively returning "no"), in its fail state (meaning that the correctness condition has been violated), or the machine diverges (i.e., never halts). A Turing machine is *correct* if its fail state is unreachable.

We can show the undecidability result using an elegant proof by contradiction. Assume P is a program that can decide whether or not any given total Turing machine is correct. (If the input to P is not a total Turing machine, P's output is unspecified – we only require it to correctly analyze Turing machines that always halt.) Let us say that P halts in its accept state if and only if the given Turing machine is correct, and it halts in the reject state otherwise. Our goal is to show that P cannot exist.

If P exists, then we can also build another Turing machine, let us call it M, that takes as input the encoding $e(T)$ of a Turing machine T and then builds the encoding $e(S_T)$ of yet another Turing machine S_T, which behaves as follows: S_T is essentially a universal Turing machine that is specialized to simulate T on input $e(T)$. Let w denote the input to S_T. Now S_T is constructed such that it simulates T on input $e(T)$ for at most $|w|$ moves. If the simulation ends in T's accept state, then S_T goes to its fail state. It is obviously possible to create S_T in such a way that this is the only way it can reach its fail state. If the simulation does not end in T's accept state (that is, $|w|$ moves have been made, or the simulation reaches T's reject or fail state), then S_T goes to its accept state or its reject state (which one we choose does not matter). This completes the explanation of how

[2] Technically, we here restrict ourselves to safety properties; liveness properties can be addressed similarly using other models of computability.

S_T works relative to T and w. Note that S_T never diverges, and it reaches its fail state if and only if T accepts input $e(T)$ after at most $|w|$ moves. After building $e(S_T)$, M passes it to our hypothetical program analyzer P. Assuming that P works as promised, it ends in accept if S_T is correct, in which case we also let M halt in its accept state, and in reject otherwise, in which case M similarly halts in its reject state.



We now ask: Does M accept input $e(M)$? That is, what happens if we run M with $T = M$? If M does accept input $e(M)$, it must be the case that P accepts input $e(S_T)$, which in turn means that S_T is correct, so its fail state is unreachable. In other words, for any input w, no matter its length, S_T does not reach its fail state. This in turn means that T does not accept input $e(T)$. However, we have $T = M$, so this contradicts our assumption that M accepts input $e(M)$. Conversely, if M rejects input $e(M)$, then P rejects input $e(S_T)$, so the fail state of S_T is reachable for some input v. This means that there must exist some w such that the fail state of S_T is reached in $|w|$ steps on input v, so T must accept input $e(T)$, and again we have a contradiction. By construction M halts in either accept or reject on any input, but neither is possible for input $e(M)$. In conclusion, the ideal program correctness analyzer P cannot exist.

Exercise 1.2: In the above proof, the hypothetical program analyzer P is only required to correctly analyze programs that always halt. How can the proof be simplified if we want to prove the following weaker property? There exists no Turing machine P that can decide whether or not the fail state is reachable in a given Turing machine. (Note that the given Turing machine is now not assumed to be total.)

Chapter 2

A Tiny Imperative Programming Language

We use a tiny imperative programming language, called *TIP*, throughout the following chapters. It is designed to have a minimal syntax and yet to contain all the constructions that make static analyses interesting and challenging. Different language features are relevant for the different static analysis concepts, so in each chapter we focus on a suitable fragment of the language.

2.1 The Syntax of TIP

In this section we present the formal syntax of the TIP language, based on context-free grammars. TIP programs interact with the world simply by reading input from a stream of integers (for example obtained from the user's keyboard) and writing output as another stream of integers (to the user's screen). The language lacks many features known from commonly used programming languages, for example, global variables, records, objects, nested functions, and type annotations. We will consider some of those features in exercises in later chapters.

Expressions

The basic expressions all denote integer values:

$I \to$ `0` | `1` | `-1` | `2` | `-2` | ...
$X \to$ `x` | `y` | `z` | ...
$E \to I$
| X
| E `+` E | E `-` E | E `*` E | E `/` E | E `>` E | E `==` E

| `(` E `)`
| `input`

Expressions E include integer constants I and variables X. The `input` expression reads an integer from the input stream. The comparison operators yield 0 for false and 1 for true. Function calls and pointer expressions will be added later.

Statements

The simple statements S are familiar:

$S \to X$ `=` E `;`
| `output` E `;`
| S S
|
| `if (`E`) {` S `}` $\left[\,$`else {` S `}`$\,\right]^?$
| `while (`E`) {` S `}`

We use the notation $\left[\ldots\right]^?$ to indicate optional parts. In the conditions we interpret 0 as false and all other values as true. The `output` statement writes an integer value to the output stream.

Functions

A function declaration F contains a function name, a number of parameters, local variable declarations, a body statement, and a return expression:

$F \to X$ `(` X`,`$\ldots$`,`X `) {` $\left[\,$`var` X`,`$\ldots$`,`X`;` $\,\right]^?$ S `return` E`; }`

The `var` block declares a collection of uninitialized local variables. Function calls are an extra kind of expression:

$E \to X$ `(` E`,`$\ldots$`,`E `)`

We sometimes treat `var` blocks and `return` instructions as statements.

Pointers

To allow dynamic memory allocation, we introduce heap pointers:

$E \to$ `alloc`
| `&`X
| `*`E
| `null`

The first expression allocates a new cell in the heap, the second expression creates a pointer to a variable (we refer to such pointers as heap pointers, even though local variables may technically reside in the call stack), and the third expression dereferences a pointer value. In order to assign values through pointers we allow another form of assignment:

$S \rightarrow$ `*`X `=` E `;`

In such an assignment, if the variable on the left-hand-side holds a pointer to a heap cell or local variable, then the value of the right-hand-side expression is stored in that cell. Note that pointers and integers are distinct values, so pointer arithmetic is not possible. It is of course limiting that `alloc` only allocates a single heap cell, but this is sufficient to illustrate the challenges that pointers impose.

We also allow another kind of pointer, namely function pointers, which makes functions first-class values. The name of a function can be used as a variable that points to the function. In order to use function pointers, we add a generalized form of function calls (sometimes called *computed* function calls, in contrast to the simple *direct* calls described earlier):

$E \rightarrow$ `(`E`)(` E`,`$\ldots$`,`E `)`

Unlike simple function calls, the function being called is now an expression (enclosed by parentheses) that evaluates to a function pointer. Function pointers serve as a primitive model for objects or higher-order functions.

Programs

A complete program is just a collection of functions:

$P \rightarrow F \ldots F$

(We sometimes also refer to indivial functions or statements as programs.) For a complete program, the function named `main` is the one that initiates execution. Its arguments are supplied in sequence from the beginning of the input stream, and the value that it returns is appended to the output stream.

To keep the presentation short, we deliberately have not specified all details of the TIP language, neither the syntax nor the semantics.

Exercise 2.1: Identify some of the under-specified parts of the TIP language, and propose meaningful choices to make it more well-defined.

2.2 Example Programs

The following TIP programs all compute the factorial of a given integer. The first one is iterative:

```
iterate(n) {
  var f;
  f = 1;
  while (n>0) {
    f = f*n;
    n = n-1;
```

```
  }
  return f;
}
```

The second program is recursive:

```
recurse(n) {
  var f;
  if (n==0) { f=1; }
  else { f=n*recurse(n-1); }
  return f;
}
```

The third program is unnecessarily complicated:

```
foo(p,x) {
  var f,q;
  if (*p==0) { f=1; }
  else {
    q = alloc;
    *q = (*p)-1;
    f=(*p)*((x)(q,x));
  }
  return f;
}
```

```
main() {
   var n;
   n = input;
   return foo(&n,foo);
}
```

2.3 Normalization

A rich and flexible syntax is useful when writing programs, but when describing and implementing static analyses, it is often convenient to work with a syntactically simpler language. For this reason we sometimes *normalize* programs by transforming them into equivalent but syntactically simpler ones. A particularly useful normalization is to flatten nested pointer expressions, such that pointer dereferences are always of the form `*`X rather than the more general `*`E, and similarly, computed function calls are always of the form `(`X`)(`X`,...,`X`)` rather than `(`E`)(`E`,...,`E`)`. It may also be useful to flatten arithmetic expressions, arguments to direct calls, branch conditions, and return expressions.

Exercise 2.2: Argue that any program can be normalized so that there are no nested expressions.

Exercise 2.3: Show how the following statement can be normalized:
`x = (**f)(g()+h());`

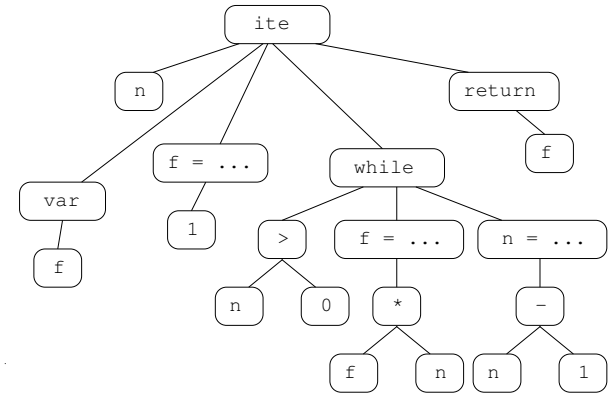
Exercise 2.4: In the current syntax for TIP, heap assignments are restricted to the form $*X = E$. Languages like C allow the more general $*E_1 = E_2$ where E_1 is an expression that evaluates to a (non-function) pointer. Explain how the statement `**x=**y;` can be normalized to fit the current TIP syntax.

TIP uses lexical scoping, however, we make the notationally simplifying assumption that all declared variable and function names are unique in a program, i.e. that no identifiers is declared more than once.

Exercise 2.5: Argue that any program can be normalized so that all declared identifiers are unique.

2.4 Abstract Syntax Trees

Abstract syntax trees (ASTs) as known from compiler construction provide a representation of programs that is suitable for flow-insensitive analysis, for example, type analysis (Chapter 3), control flow analysis (Chapter 8), and pointer analysis (Chapter 9). Such analyses ignore the execution order of statements in a function or block, which makes ASTs a convenient representation. As an example, the AST for the `ite` program can be illustrated as follows.



With this representation, it is easy to extract the set of statements and and their structure for each function in the program.

2.5 Control Flow Graphs

For flow-sensitive analysis , in particular dataflow analysis (Chapter 5), where statement order matters it is more convenient to view the program as a *control flow graph*, which is a different representation of the program source.

We first consider the subset of the TIP language consisting of a single function body without pointers. Control flow graphs for programs comprising multiple functions are treated in Chapters 7 and 8.

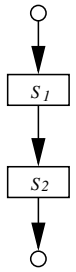
A control flow graph (CFG) is a directed graph, in which *nodes* correspond to statements and *edges* represent possible flow of control. For convenience, and without loss of generality, we can assume a CFG to always have a single point of entry, denoted *entry*, and a single point of exit, denoted *exit*. We may think of these as no-op statements.

If v is a node in a CFG then $pred(v)$ denotes the set of predecessor nodes and $succ(v)$ the set of successor nodes.

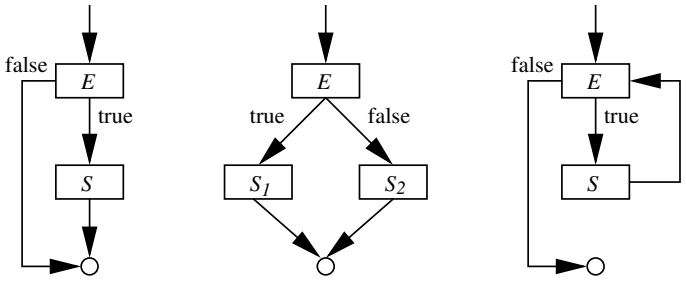
Control Flow Graphs for Statements

For now, we only consider simple statements, for which CFGs may be constructed in an inductive manner. The CFGs for assignments, `output`, `return` statements, and declarations look as follows:

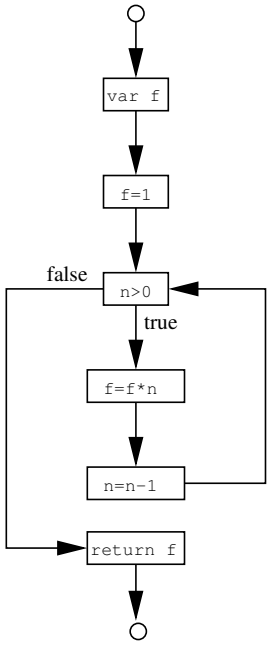
For the sequence S_1 S_2, we eliminate the exit node of S_1 and the entry node of S_2 and glue the statements together:



Similarly, the other control structures are modeled by inductive graph constructions (sometimes with branch edges labeled with true and false):



Using this systematic approach, the iterative factorial function results in the following CFG:



Exercise 2.6: Draw the AST and the CFG for the `rec` program from Section 2.2.

Exercise 2.7: If TIP were to be extended with a `do-while` construct (as in `do { x=x-1; } while(x>0)`), what would the corresponding control flow graphs look like?

Chapter 3

Type Analysis

The TIP programming language does not have explicit type declarations, but of course the various operations are intended to be applied only to certain arguments. Specifically, the following restrictions seem reasonable:

- arithmetic operations and comparisons apply only to integers;
- conditions in control structures must be integers;
- only integers can be input and output of the main function;
- only functions can be called, and with correct number of arguments; and
- the unary * operator only applies to heap pointers.

We assume that their violation results in runtime errors. Thus, for a given program we would like to know that these requirements hold during execution. Since this is an nontrivial question, we immediately know (Section 1.1) that it is undecidable.

We resort to a conservative approximation: *typability*. A program is typable if it satisfies a collection of type constraints that is systematically derived, typically from the program AST. The type constraints are constructed in such a way that the above requirements are guaranteed to hold during execution, but the converse is not true. Thus, our type checker will be conservative and reject some programs that in fact will not violate any requirements during execution.

Exercise 3.1: Type checking also in mainstream languages like Java may reject programs that cannot encounter runtime type errors. Give an example of such a program. To make the exercise more interesting, every instruction in your program should be reachable by some input.

Exercise 3.2: Even popular programming languages may have static type systems that are unsound. Inform yourself about Java's covariant typing of arrays. Construct an example Java program that passes all of `javac`'s type checks but generates a runtime error due to this covariant typing. (Note that, because you do receive runtime errors, Java's *dynamic* type system *is* sound, which is important to avert malicious attacks, e.g. through type confusion or memory corruption.)

3.1 Types

We first define a language of *types* that will describe possible values:

$$\begin{aligned}\tau \rightarrow\ & \texttt{int} \\ \mid\ & \texttt{\&}\tau \\ \mid\ & \texttt{(}\tau\texttt{,...,}\tau\texttt{)->}\tau\end{aligned}$$

These type terms describe respectively integers, heap pointers, and functions. As an example, we can assign the type `(int)->int` to the `iterate` function from Section 2.2 and the type `&int` to the first parameter `p` of the `foo` function. Each kind of term is characterized by a *term constructor* with some arity. For example, `&` is a term constructor with arity 1 as it has one sub-term, and the arity of a function type constructor `(...)->...` is the number of function parameters plus one for the return type.

The grammar would normally generate finite types, but for recursive functions and data structures we need *regular* types. Those are defined as regular trees defined over the above constructors. Recall that a possibly infinite tree is regular if it contains only finitely many different subtrees.

For example, we need infinite types to describe the type of the `foo` function from Section 2.2, since the second parameter `x` may refer to the `foo` function itself:

```
(&int,(&int,(&int,(&int,...)->int)->int)->int)->int
```

To express such recursive types consisely, we add the μ operator and type variables α to the language of types:

$$\begin{aligned}\tau \rightarrow\ & \mu\alpha.\tau \\ \mid\ & \alpha \\ \alpha \rightarrow\ & x_1 \mid x_2 \mid \ldots\end{aligned}$$

A type of the form $\mu\alpha.\tau[\alpha]$ is considered identical to the type $\tau[\mu\alpha.\tau/\alpha]$.[1] With this extra notation, the type of the `foo` function can be expressed like this:

[1] Think of a term $\mu\alpha.\tau$ as a quantifier that binds the type variable α in the sub-term τ. An occurrence of α in a term τ is *free* if it is not bound by an enclosing μ. The notation $\tau_1[\tau_2/\alpha]$ denotes a copy of τ_1 where all free occurrences of α have been substituted by τ_2.

μx_1.(&int,x_1)->int

Exercise 3.3: Show how regular types can be represented by finite automata so that two types are equal if their automata accept the same language.

We allow type variables not to be bound by an enclosing μ. Such type variables are implicitly universally quantified, meaning that they represent any type. Consider for example the following function:

```
store(x,y) {
  *y = x;
  return 0;
}
```

It has type (τ_1,&τ_1)->int for any type τ_1, which corresponds to the polymorphic behavior it displays. Note that such type variables are not necessarily entirely unconstrained: the type of `x` may be anything, but it must match the type of whatever `y` points to. The more restricted type (int,&int)->int is also a valid type for the `store` function, but we are usually interested in the most general solutions.

Exercise 3.4: What are the types of `rec`, `f`, and `n` in the recursive factorial program from Section 2.2?

Exercise 3.5: Write a TIP program that contains a function with type ((int)->int)->(int,int)->int.

Type variables are not only useful for expressing recursive types; we also use them in the following section to express systems of type constraints.

3.2 Type Constraints

For a given program we generate a constraint system and define the program to be typable when the constraints are solvable. In our case we only need to consider equality constraints over regular type terms with variables. This class of constraints can be efficiently solved using a unification algorithm.

For each identifier (i.e. local variable, function parameter, or function name) X we introduce a type variable $[\![X]\!]$, and for each occurrence of a non-identifier expression E a type variable $[\![E]\!]$. Here, E refers to a concrete node in the abstract syntax tree—not to the syntax it corresponds to. This makes our notation slightly ambiguous but simpler than a pedantically correct description. (To avoid ambiguity, one could, for example, use the notation $[\![E]\!]_v$ where v is a unique ID of the syntax tree node.) Assuming that all declared identifiers are unique (see Exercise 2.5), there is no need to use different type variables for different occurrences of the same identifier.

The constraints are systematically defined for each construct in our language:

I:	$[\![I]\!] = \texttt{int}$
E_1 op E_2:	$[\![E_1]\!] = [\![E_2]\!] = [\![E_1 \text{ op } E_2]\!] = \texttt{int}$
E_1==E_2:	$[\![E_1]\!] = [\![E_2]\!] \ \wedge \ [\![E_1==E_2]\!] = \texttt{int}$
input:	$[\![\texttt{input}]\!] = \texttt{int}$
$X = E$:	$[\![X]\!] = [\![E]\!]$
output E:	$[\![E]\!] = \texttt{int}$
if (E) S:	$[\![E]\!] = \texttt{int}$
if (E) S_1 else S_2:	$[\![E]\!] = \texttt{int}$
while (E) S:	$[\![E]\!] = \texttt{int}$
$X(X_1,\ldots,X_n)\{\ldots\texttt{return}\ E;\}$:	$[\![X]\!] = ([\![X_1]\!],\ldots,[\![X_n]\!])\texttt{->}[\![E]\!]$
$X(E_1,\ldots,E_n)$:	$[\![X]\!] = ([\![E_1]\!],\ldots,[\![E_n]\!])\texttt{->}[\![X(E_1,\ldots,E_n)]\!]$
$(E)(E_1,\ldots,E_n)$:	$[\![E]\!] = ([\![E_1]\!],\ldots,[\![E_n]\!])\texttt{->}[\![(E)(E_1,\ldots,E_n)]\!]$
&X:	$[\![\&X]\!] = \&[\![X]\!]$
alloc:	$[\![\texttt{alloc}]\!] = \&\alpha$
null:	$[\![\texttt{null}]\!] = \&\alpha$
*E:	$[\![E]\!] = \&[\![*E]\!]$
*X = E:	$[\![X]\!] = \&[\![E]\!]$

In the above rules, each occurrence of α denotes a fresh type variable. Note that variable references and declarations do not yield any constraints and that parenthesized expression are not present in the abstract syntax.

For the program

```
short() {
  var x, y, z;
  x = input;
  y = alloc;
  *y = x;
  z = *y;
  return z;
}
```

we obtain the following constraints:

$[\![\texttt{short}]\!] = \texttt{()->}[\![\texttt{z}]\!]$
$[\![\texttt{input}]\!] = \texttt{int}$
$[\![\texttt{x}]\!] = [\![\texttt{input}]\!]$
$[\![\texttt{alloc}]\!] = \&x_1$
$[\![\texttt{y}]\!] = [\![\texttt{alloc}]\!]$
$[\![\texttt{y}]\!] = \&[\![\texttt{x}]\!]$
$[\![\texttt{z}]\!] = [\![\texttt{*y}]\!]$
$[\![\texttt{y}]\!] = \&[\![\texttt{*y}]\!]$

Most of the constraint rules are straightforward. For example, for any syntactic occurrence of E_1==E_2 in the program being analyzed, the two sub-expressions E_1 and E_2 must have the same type, and the result is always of type integer.

Exercise 3.6: Explain each of the above type constraint rules, most importantly those involving functions and heap pointers.

All term constructors furthermore satisfy the general term equality axiom:

$$c(t_1, \ldots, t_n) = c'(t'_1, \ldots, t'_n) \;\Rightarrow\; t_i = t'_i \text{ for each } i$$

where c and c' are term constructors and each t_i and t'_i is a sub-term. In the previous example two of the constraints are ⟦`y`⟧ = `&`⟦`x`⟧and ⟦`y`⟧ = `&`⟦`*y`⟧, so by the term equality axiom we also have ⟦`x`⟧= ⟦`*y`⟧.

In this way, a given program gives rise to a collection of equality constraints on type terms with variables, and the collection of constraints can be built by a simple traversal of the AST of the program being analyzed.

A *solution* assigns to each type variable a type, such that all equality constraints are satisfied. The correctness claim for the type analysis is that the existence of a solution implies that the specified runtime errors cannot occur during execution. A solution for the `short` program is the following:

⟦`short`⟧ = `()->int`
⟦`x`⟧ = `int`
⟦`y`⟧ = `&int`
⟦`z`⟧ = `int`

Exercise 3.7: Assume `y = alloc` in the `short` function is changed to `y = 42`. Show that the resulting constraints are unsolvable.

Exercise 3.8: Extend TIP with *procedures*, which, unlike functions, do not return anything. Show how to extend the language of types and the type constraint rules accordingly.

3.3 Solving Constraints with Unification

If solutions exist, then they can be computed in almost linear time using a unification algorithm for regular terms as explained below. Since the constraints may also be extracted in linear time, the whole type analysis is quite efficient.

The unification algorithm is based on the familiar union-find data structure (also called a disjoint-set data structure) for representing and manipulating equivalence relations. This data structure consists of a directed graph of nodes that each have exactly one edge to its *parent* node (which may be the node itself in which case it is called a *root*). Two nodes are *equivalent* if they have a common ancestor, and each root is the *canonical representative* of its equivalence class. Three operations are provided:[2]

[2]We here consider a simple version of union-find without union-by-rank; for a description of the full version with almost-linear worst case time complexity see a textbook on data structures.

- MAKESET(x): adds a new node x that initially is its own parent.
- FIND(x): finds the canonical representative of x by traversing the path to the root, performing path compression on the way (meaning that the parent of each node on the traversed path is set to the canonical representative).
- UNION(x,y): finds the canonical representatives of x and y, and makes one parent of the other unless they are already equivalent.

In pseudo-code:

```
procedure MAKESET(x)
    x.parent := x
end procedure

procedure FIND(x)
    if x.parent ≠ x then
        x.parent := FIND(x.parent)
    end if
    return x.parent
end procedure

procedure UNION(x, y)
    x^r := FIND(x)
    y^r := FIND(y)
    if x^r ≠ y^r then
        x^r.parent := y^r
    end if
end procedure
```

The unification algorithm uses union-find by associating a node with each term (including sub-terms) in the constraint system. For each term τ we initially invoke MAKESET(τ). Note that each term at this point is either a type variable or a proper type (i.e. integer, heap pointer, or function); μ terms are only produced for presenting solutions to constraints, as explained below. For each constraint $\tau_1 = \tau_2$ we invoke UNIFY(τ_1, τ_2), which unifies the two terms if possible and enforces the general term equality axiom by unifiying sub-terms recursively:

```
procedure UNIFY(τ1,τ2)
    τ1^r := FIND(τ1)
    τ2^r := FIND(τ2)
    if τ1^r ≠ τ2^r then
        if τ1^r and τ2^r are both type variables then
            UNION(τ1^r, τ2^r)
        else if τ1^r is a type variable and τ2^r is a proper type then
            UNION(τ1^r, τ2^r)
        else if τ1^r is a proper type and τ2^r is a type variable then
```

```
            UNION(τ_2^r, τ_1^r)
        else if τ_1^r and τ_2^r are proper types with same type constructor then
            UNION(τ_1^r, τ_2^r)
            for each pair of sub-terms τ_1' and τ_2' of τ_1^r and τ_2^r, respectively do
                UNIFY(τ_1', τ_2')
            end for
        else
            unification failure
        end if
    end if
end procedure
```

Unification fails if attempting to unify two terms with different constructors (where function constructors are considered different if they have different arity).

Note that the UNION(x, y) operation is asymmetric: it always picks the canonical representative of the resulting equivalence class as the one from equivalence class of the second argument y. Also, UNIFY is carefully constructed such that the second argument to UNION can only be a type variable if the first argument is also a type variable. This means that proper types take precedence over type variables for becoming canonical representatives, which makes it easier to read off the solution after all constraints have been processed: For each type variable, simply invoke FIND to find the canonical representative of its equivalence class. The only complication arises if the canonical representative is itself a type variable, in which case the desired type is recursive, so we introduce a μ term accordingly.

Exercise 3.9: Argue that the unification algorithm works correctly, in the sense that it finds a solution to the given constraints if one exists. Additionally, argue that if multiple solutions exist, the algorithm finds the uniquely most general one.

The complicated factorial program from Section 2.2 generates the following constraints (duplicates omitted):

⟦`foo`⟧ = (⟦`p`⟧,⟦`x`⟧)->⟦`f`⟧	⟦`*p==0`⟧ = `int`
⟦`*p`⟧ = int	⟦`f`⟧ = ⟦`1`⟧
⟦`1`⟧ = `int`	⟦`0`⟧ = `int`
⟦`p`⟧ = &⟦`*p`⟧	⟦`q`⟧ = ⟦`alloc`⟧
⟦`alloc`⟧ = &α	⟦`q`⟧ = &⟦`(*p)-1`⟧
⟦`q`⟧ = &⟦`*q`⟧	⟦`*p`⟧ = `int`
⟦`f`⟧ = ⟦`(*p)*((x)(q,x))`⟧	⟦`(*p)*((x)(q,x))`⟧ = `int`
⟦`(x)(q,x)`⟧ = `int`	⟦`x`⟧ = (⟦`q`⟧,⟦`x`⟧)->⟦`(x)(q,x)`⟧
⟦`input`⟧ = `int`	⟦`main`⟧ = ()->⟦`foo(&n,foo)`⟧
⟦`n`⟧ = ⟦`input`⟧	⟦`&n`⟧ = &⟦`n`⟧
⟦`foo`⟧ = (⟦`&n`⟧,⟦`foo`⟧)->⟦`foo(&n,foo)`⟧	⟦`*p`⟧ = ⟦`0`⟧

These constraints have a solution, where most variables are assigned `int`, except

these:

$[\![\texttt{p}]\!] = \texttt{\&int}$
$[\![\texttt{q}]\!] = \texttt{\&int}$
$[\![\texttt{alloc}]\!] = \texttt{\&int}$
$[\![\texttt{x}]\!] = \mu x_1.\texttt{(\&int,}x_1\texttt{)->int}$
$[\![\texttt{foo}]\!] = \mu x_1.\texttt{(\&int,}x_1\texttt{)->int}$
$[\![\texttt{\&n}]\!] = \texttt{\&int}$
$[\![\texttt{main}]\!] = \texttt{()->int}$

As mentioned in Section 3.1, recursive types are needed for the `foo` function and the `x` parameter. Since a solution exists, we conclude that our program is type correct.

Exercise 3.10: Check (by hand or using the Scala implementation) that the constraints and the solution shown above are correct for the complicated factorial program.

Recursive types are also required when analyzing TIP programs that manipulate data structures. The example program

```
var p;
p = alloc;
*p = p;
```

creates these constraints:

$[\![\texttt{p}]\!] = \texttt{\&}x_1$
$[\![\texttt{p}]\!] = \texttt{\&}[\![\texttt{p}]\!]$

which has the solution $[\![\texttt{p}]\!] = \mu x_2.\texttt{\&}x_2$ that can be unfolded to $[\![\texttt{p}]\!] = \texttt{\&\&\&\&\&\&\&}\ldots$.

Exercise 3.11: Generate and solve the constraints for the `iterate` example program from Section 2.2.

Exercise 3.12: Generate and solve the type constraints for this program:

```
map(l,f,z) {
  var r;
  if (l==null) r=z;
  else r=(f)(map(*l,f,z));
  return r;
}

foo(i) {
  return i+1;
}

main() {
  var h,t,n;
  t = null;
  n = 42;
  while (n>0) {
    n = n-1;
    h = alloc;
    *h = t;
    t = h;
  }
  return map(h,foo,0);
}
```

What is the output from running the program?
(Try to find the solutions manually; you can then use the Scala implementation to check that they are correct.)

3.4 Limitations of the Type Analysis

The type analysis is of course only approximate, which means that certain programs will be unfairly rejected. A simple example is this:

```
f() {
  var x;
  x = alloc;
  x = 42;
  return x + 87;
}
```

This program has no type errors at runtime, but it is rejected by our type checker because the analysis is *flow-insensitive*: the order of execution of the program instructions is abstracted away by the analysis, so intuitively it does not know

that `x` must be an integer at the return expression. In the following chapters we shall see how to perform *flow-sensitive* analysis that does distinguish between the different program points.

Another example is

```
bar(g,x) {
  var r;
  if (x==0) { r=g; } else { r=bar(2,0); }
  return r+1;
}

main() {
  return bar(null,1);
}
```

which never causes an error but is not typable since it among others generates constraints equivalent to

$$\texttt{int} = [\![\texttt{r}]\!] = [\![\texttt{g}]\!] = \&\alpha$$

which are clearly unsolvable.

Exercise 3.13: Explain the runtime behavior of this program, and why it is unfairly rejected by our type analysis.

It is possible to use a more powerful polymorphic type analysis to accept the above program, but infinitely many other examples will inevitably remain rejected.

Another problem is that this type system ignores several other runtime errors, such as dereference of `null` pointers, reading of uninitialized variables, division by zero, and the more subtle *escaping stack cell* demonstrated by this program:

```
baz() {
  var x;
  return &x;
}

main() {
  var p;
  p = baz();
  *p = 1;
  return *p;
}
```

The problem is that `*p` denotes a stack cell that has *escaped* from the `baz` function. As we shall see in the following chapters, these problems can instead be handled by other kinds of static analysis.

Exercise 3.14: We extend the TIP language with array operations. Array values are constructed using a new form of expressions:

$E \to$ `{` E_1 `,` E_2 `,` $\ldots$ `,` E_k `}`

(for $k \geq 0$), and individual elements are read and written as follows:

$E \to E_1$ `[` E_2 `]`
$S \to E_1$ `[` E_2 `]` `=` E_3

The type system is extended accordingly with an array type constructor:

$\tau \to \tau$ `[]`

Give appropriate type constraints for array operations. Then use the type analysis to check that the following program is typable and infer the type of each variable:

```
var x,y,z,t;
x = {2,4,8,16,32,64};
y = x[x[3]];
z = {{},x};
t = z[1];
t[2] = y;
```

Chapter 4

Lattice Theory

The technique for static analysis that we will study next is based on the mathematical theory of *lattices*, which we briefly review in this chapter.

4.1 Motivating Example: Sign Analysis

As a motivating example, assume that we wish to design an analysis that can find out the possible signs of the integer values of variables and expressions in a given program. In concrete executions, values can be arbitrary integers. In contrast, our analysis considers an abstraction of the integer values by grouping them into the three categories, or *abstract values*: positive (+), negative (-), and zero (`0`). Similar to the analysis we considered in Chapter 3, we circumvent undecidability by introducing approximation. That is, the analysis must be prepared to handle uncertain information, in this case situations where it does not know the sign of some expression, so we add a special abstract value (?) representing "don't know". We must also decide what information we are interested in for the cases where the sign of some expression is, for example, positive in some executions but not in others. For this example, let us assume we are interested in *definite* information, that is, the analysis should only report + for a given expression if it is certain that this expression will evaluate to a positive number in *every* execution of that expression and ? otherwise. In addition, it turns out to be beneficial to also introduce an abstract value $\bot$ for expressions whose values are not numbers (but instead, say, pointers) or have no value in any execution because they are unreachable from the program entry.

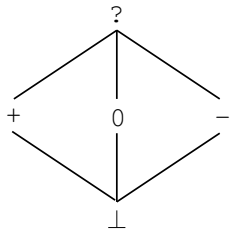
Consider this program:

```
var a,b,c;
a = 42;
b = 87;
```

```
if (input) {
  c = a + b;
} else {
  c = a - b;
}
```

Here, the analysis could conclude that `a` and `b` are positive numbers in all possible executions at the end of the program. The sign of `c` is either positive or negative depending on the concrete execution, so the analysis must report `?` for that variable.

For this analysis we have an *abstract domain* consisting of the five abstract values $\{+, -, 0, ?, \bot\}$, which we can organize as follows with the least precise information at the top and the most precise information at the bottom:



The ordering reflects the fact that $\bot$ represents the empty set of integer values and `?` represents the set of all integer values. Note that `?` may arise for different reasons: (1) In the example above, there exist executions where `c` is positive and executions where `c` is negative, so, for this choice of abstract domain, `?` is the only sound option. (2) Due to undecidability, imperfect precision is inevitable, so no matter how we design the analysis there will be programs where, for example, some variable can only have a positive value in any execution but the analysis is not able to show that it could not also have a negative value (recall the `TM(j)` example from Chapter 1).

The five-element abstract domain shown above is an example of a so-called lattice. We continue the development of the sign analysis in Section 5.1, but we first need the mathematical foundation in place.

4.2 Lattices

A *partial order* is a set S equipped with a binary relation $\sqsubseteq$ where the following conditions are satisfied:

- reflexivity: $\forall x \in S : x \sqsubseteq x$
- transitivity: $\forall x, y, z \in S : x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z$
- anti-symmetry: $\forall x, y \in S : x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y$

When $x \sqsubseteq y$ we say that y is a *safe approximation* of x, or that x is *at least as precise* as y. Formally, a lattice is a pair $(S, \sqsubseteq)$, but we sometimes use the same name for the lattice and its underlying set.

Let $X \subseteq S$. We say that $y \in S$ is an *upper bound* for X, written $X \sqsubseteq y$, if we have $\forall x \in X : x \sqsubseteq y$. Similarly, $y \in S$ is a *lower bound* for X, written $y \sqsubseteq X$, if $\forall x \in X : y \sqsubseteq x$. A *least* upper bound, written $\sqcup X$, is defined by:

$$X \sqsubseteq \sqcup X \ \wedge \ \forall y \in S : X \sqsubseteq y \Rightarrow \sqcup X \sqsubseteq y$$

Dually, a *greatest* lower bound, written $\sqcap X$, is defined by:

$$\sqcap X \sqsubseteq X \ \wedge \ \forall y \in S : y \sqsubseteq X \Rightarrow y \sqsubseteq \sqcap X$$

For pairs of elements, we sometimes use the infix notation $x \sqcup y$ instead of $\sqcup\{x, y\}$ and $x \sqcap y$ instead of $\sqcap\{x, y\}$.

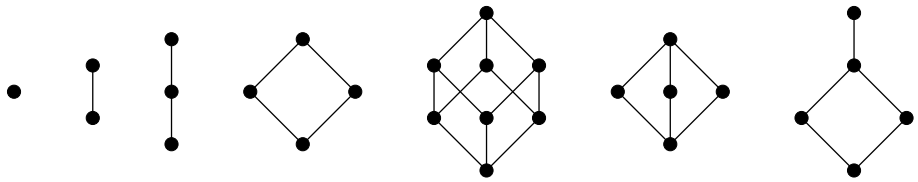
Exercise 4.1: Prove that if $\bigsqcup X$ exists, then it must be unique.

Exercise 4.2: Prove that if $x \sqcup y$ exists then $x \sqsubseteq y \Leftrightarrow x \sqcup y = y$, and conversely, if $x \sqcap y$ exists then $x \sqsubseteq y \Leftrightarrow x \sqcap y = x$.

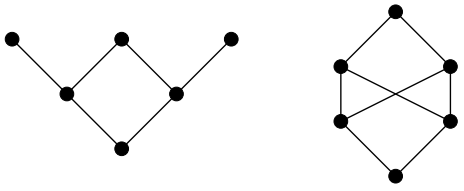
A *lattice* is a partial order in which $\sqcup X$ and $\sqcap X$ exist for all $X \subseteq S$.[1]

Exercise 4.3: Argue that the abstract domain presented in Section 4.1 is indeed a lattice.

Any finite partial order may be illustrated by a Hasse diagram in which the elements are nodes and the order relation is the transitive closure of edges leading from lower to higher nodes. With this notation, all of the following partial orders are also lattices:



whereas these partial orders are *not* lattices:



[1] This definition of a lattice we use here is typically called a *complete* lattice in the literature, but we choose to use the shorter name. Also, for our purposes it often suffices to consider *join semi-lattices* which have $\sqcup X$ but not necessarily $\sqcap X$ for all $X \subseteq S$. Many lattices are *finite*. For those the lattice requirements reduce to observing that $\bot$ and $\top$ exist and that every pair of elements x and y have a least upper bound $x \sqcup y$ and a greatest lower bound $x \sqcap y$.

Exercise 4.4: Why do these two diagrams not define lattices?

Every lattice has a unique *largest* element denoted $\top$ and a unique *smallest* element denoted $\bot$.

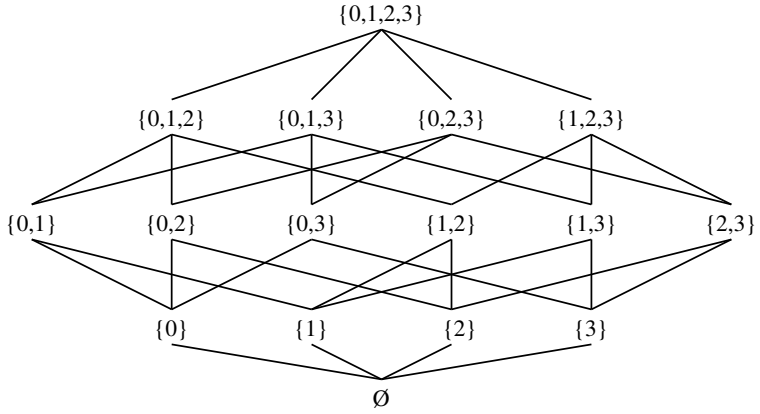
Exercise 4.5: Prove that $\sqcup S$ and $\sqcap S$ are the unique largest element and the unique smallest element, respectively, in S. In other words, we have $\top = \sqcup S$ and $\bot = \sqcap S$.

Exercise 4.6: Prove that $\sqcup S = \sqcap \emptyset$ and that $\sqcap S = \sqcup \emptyset$.

The *height* of a lattice is defined to be the length of the longest path from $\bot$ to $\top$. As an example, the height of the sign analysis lattice from Section 4.1 is 2. For some lattices the height is infinite.

4.3 Constructing Lattices

Every finite set A defines a lattice $(2^A, \subseteq)$, where $\bot = \emptyset$, $\top = A$, $x \sqcup y = x \cup y$, and $x \sqcap y = x \cap y$. We call this the *powerset lattice* for A. For a set with four elements, the powerset lattice looks like this:



The above powerset lattice has height 4. In general, the lattice $(2^A, \subseteq)$ has height $|A|$.

If $L_1, L_2, \ldots, L_n$ are lattices, then so is the *product*:

$$L_1 \times L_2 \times \ldots \times L_n = \{(x_1, x_2, \ldots, x_n) \mid x_i \in L_i\}$$

where the lattice order $\sqsubseteq$ is defined pointwise:[2]

$$(x_1, x_2, \ldots, x_n) \sqsubseteq (x'_1, x'_2, \ldots, x'_n) \quad \Leftrightarrow \quad \forall i = 1, 2, \ldots, n : x_i \sqsubseteq x'_i$$

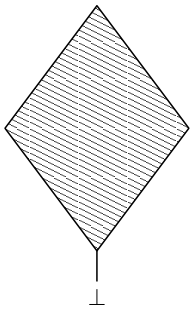
[2]We often abuse notation by using the same symbol $\sqsubseteq$ for many different order relations, in this case from the $n + 1$ different lattices, but it should always be clear from the context which lattice it belongs to. The same applies to the other operators $\sqsupseteq$, $\sqcup$, $\sqcap$ and the top/bottom symbols $\top$, $\bot$.

Products of n identical lattices may be written consisely as $L^n = \underbrace{L \times L \times \ldots \times L}_{n}$.

Exercise 4.7: Show that the $\sqcup$ and $\sqcap$ operators for a product lattice $L_1 \times L_2 \times \ldots \times L_n$ can be computed pointwise (i.e. in terms of the $\sqcup$ and $\sqcap$ operators from L_1, L_2, …, L_k).

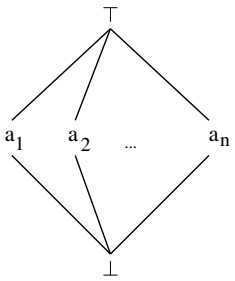
Exercise 4.8: Show that $height(L_1 \times \ldots \times L_n) = height(L_1) + \ldots + height(L_n)$.

If L is a lattice, then so is $lift(L)$, which is a copy of L but with a new bottom element:



It has $height(lift(L)) = height(L) + 1$ if L has finite height.

If A is a set (not necessarily a lattice), then $flat(A)$ illustrated by



is a lattice with height 2.

Finally, if A is a set and L is a lattice, then we obtain a *map* lattice consisting of the set of functions from A to L, ordered pointwise:[3]

$$A \to L = \left\{ [a_1 \mapsto x_1, a_2 \mapsto x_2, \ldots] \;\middle|\; A = \{a_1, a_2, \ldots\} \wedge x_1, x_2, \ldots \in L \right\}$$

$$f \sqsubseteq g \Leftrightarrow \forall a_i \in A : f(a_i) \sqsubseteq g(a_i) \text{ where } f, g \in A \to L$$

Exercise 4.9: Show that the $\sqcup$ and $\sqcap$ operators for a map lattice $A \to L$ can be computed pointwise (i.e. in terms of the $\sqcup$ and $\sqcap$ operators from L).

[3]The notation $[a_1 \mapsto x_1, a_2 \mapsto x_2, \ldots]$ means the function that maps a_1 to x_1, a_2 to x_2, etc.

Exercise 4.10: Show that if A is finite and L has finite height then the height of the map lattice $A \to L$ is $height(A \to L) = |A| \cdot height(L)$.

If L_1 and L_2 are lattices, then a function $f : L_1 \to L_2$ is a *homomorphism* if $\forall x, y \in L_1 : f(x \sqcup y) = f(x) \sqcup f(y) \wedge f(x \sqcap y) = f(x) \sqcap f(y)$. A bijective homomorphism is called an *isomorphism*. Two lattices are *isomorphic* if there exists an isomorphism from one to the other.

Exercise 4.11: Argue that every product lattice L^n is isomorphic to a map lattice $A \to L$ for some choice of A, and vice versa.

We have already seen that the set $Sign = \{+, -, \mathtt{0}, ?, \bot\}$ with the ordering described in Section 4.1 forms a lattice that we use for describing abstract values in the sign analysis. An example of a map lattice is $StateSigns = Vars \mapsto Sign$ where $Vars$ is the set of variable names occurring in the program that we wish to analyze. Elements of this lattice describe abstract states that provide abstract values for all variables. An example of a product lattice is $ProgramSigns = StateSigns^n$ where n is the number of nodes in the CFG of the program. We shall use this lattice, which can describe abstract states for all nodes of the program CFG, in Section 5.1 for building a flow-sensitive sign analysis. Note that by Exercise 4.11 the lattice $StateSigns^n$ is isomorphic to $Nodes \to StateSigns$ where $Nodes$ is the set of CFG nodes, so which of the two variants we use when describing the sign analysis is only a matter of preferences. This example also illustrates that the lattices we use may depend on the program being analyzed: the sign analysis depends on the set of variables that occur in the program and also on its CFG nodes.

4.4 Equations, Monotonicity, and Fixed-Points

Continuing the sign analysis from Section 4.1, what are the signs of the variables at each line of the following simple program?

```
var a,b;       // 1
a = 42;        // 2
b = a + input; // 3
a = a - b;     // 4
```

We can derive a system of equations with one constraint variable for each program variable and line number from the program:[4]

$a_1 = ?$
$b_1 = ?$
$a_2 = +$

[4]We use the term *constraint variable* to denote variables that appear in mathematical constraint systems, to avoid confusion with *program variables* that appear in TIP programs.

$$
\begin{aligned}
b_2 &= b_1 \\
a_3 &= a_2 \\
b_3 &= a_2 \texttt{ + ?} \\
a_4 &= a_3 \texttt{ - } b_3 \\
b_4 &= b_3
\end{aligned}
$$

For example, a_2 denotes the abstract value of a at the program point immediately after line 2. The operators + and - here work on abstract values, which we return to in Section 5.1. In this constraint system, the constraint variables have values from the abstract value lattice $Sign$ defined in Section 4.3. We can alternatively derive the following equivalent constraint system where each constraint variable instead has a value from the abstract state lattice $StateSigns$ from Section 4.3:[5]

$$
\begin{aligned}
x_1 &= [\texttt{a} \mapsto \texttt{?}, \texttt{b} \mapsto \texttt{?}] \\
x_2 &= x_1[\texttt{a} \mapsto \texttt{+}] \\
x_3 &= x_2[\texttt{b} \mapsto x_2(\texttt{a}) \texttt{ + ?}] \\
x_4 &= x_3[\texttt{a} \mapsto x_3(\texttt{a}) \texttt{ - } x_3(\texttt{b})]
\end{aligned}
$$

Here, each constraint variable models the abstract state at a program point; for example, x_1 models the abstract state at the program point immediately after line 1. Notice that each equation only depends on preceding ones for this example program, so in this case the solution can be found by simple substition. However, mutually recursive equations may appear, for example for programs that contain loops (see Section 5.1).

Exercise 4.12: Give a solution to the constraint system above (that is, values for $x_1, \ldots, x_4$ that satisfy the four equations).

Exercise 4.13: Why is the unification solver from Chapter 3 not suitable for this kind of constraints?

We now show how to solve such constraint systems in a general setting.

A function $f : L_1 \to L_2$ where L_1 and L_2 are lattices is *monotone* (or *order-preserving*) when $\forall x, y \in L_1 : x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$. As the lattice order when used in program analysis represents precision of information, the intuition of monotonicity is that "more precise input does not result in less precise output".

[5]The notation $f[a_1 \mapsto x_n, \ldots, a_n \mapsto x_n]$ means the function that maps a_i to x_i, for each $i = 1, \ldots, n$ and for all other inputs gives the same output as the function f.

Exercise 4.14: A function $f : L \to L$ where L is a lattice is *extensive* when $\forall x \in L : x \sqsubseteq f(x)$. Assume L is the powerset lattice $2^{\{0,1,2,3,4\}}$ Give examples of different functions $L \to L$ that are, respectively,

(a) extensive and monotone,

(b) extensive but not monotone,

(c) not extensive but monotone, and

(d) not extensive and not monotone.

Exercise 4.15: Prove that every constant function is monotone.

Exercise 4.16: A function $f : L_1 \to L_2$ where L_1 and L_2 are lattices is *distributive* when $\forall x, y \in L_1 : f(x) \cup f(y) = f(x \sqcup y)$.

(a) Show that every distributive function is also monotone.

(b) Show that not every monotone function is also distributive.

Exercise 4.17: Prove that a function $f : L_1 \to L_2$ where L_1 and L_2 are lattices is monotone if and only if $\forall x, y \in L_1 : f(x) \sqcup f(y) \sqsubseteq f(x \sqcup y)$.

Exercise 4.18: Prove that function composition preserves monotonicity. That is, if $f : L_1 \to L_2$ and $g : L_2 \to L_3$ are monotone, then so is their composition $g \circ f$, which is defined by $(g \circ f)(x) = g(f(x))$.

The definition of monotonicity generalizes naturally to functions with multiple arguments: for example, a function with two arguments $f : L_1 \times L_2 \to L_3$ where L_1, L_2, and L_3 are lattices is monotone when $\forall x_1, y_1 \in L_1, x_2 \in L_2 : x_1 \sqsubseteq y_1 \Rightarrow f(x_1, x_2) \sqsubseteq f(y_1, x_2)$ and $\forall x_1 \in L_1, x_2, y_2 \in L_2 : x_2 \sqsubseteq y_2 \Rightarrow f(x_1, x_2) \sqsubseteq f(x_1, y_2)$.

Exercise 4.19: The operators $\sqcup$ and $\sqcap$ can be viewed as functions. For example, $\sqcup\{x_1, x_2\}$ where $x_1, x_2 \in L$ returns an element from L. Show that $\sqcup$ and $\sqcap$ are monotone.

Exercise 4.20: Let $f : L^n \to L^n$ be a function n arguments over a lattice L. We can view such a function in different ways: either as function with n arguments from L, or as a function with single argument from the product lattice L^n. Argue that this does not matter for the definition of monotonicity.

Exercise 4.21: Show that set difference, $X \backslash Y$, as a function with two arguments over a powerset lattice is monotone in the first argument X but not in the second argument Y.

Exercise 4.22: Recall that $f[a \mapsto x]$ denotes the function that is identical to f except that it maps a to x. Assume $f : L_1 \to (A \to L_2)$ and $g : L_1 \to L_2$ are monotone functions where L_1 and L_2 are lattices and A is a set, and let $a \in A$. (Note that the codomain of f is a map lattice.) Show that the function $h : L_1 \to (A \to L_2)$ defined by $h(x) = f(x)[a \mapsto g(x)]$ is monotone.

Also show that the following claim is *wrong*: The map update operation preserves monotonicity in the sense that if $f : L \to L$ is monotone then so is $f[a \mapsto x]$ for any lattice L and $a, x \in L$.

We say that $x \in L$ is a *fixed-point* for f if $f(x) = x$. A *least* fixed-point x for f is a fixed-point for f where $x \sqsubseteq y$ for every fixed-point y for f.

Let L be a lattice. An *equation system* over L is of the form

$$\begin{aligned} x_1 &= f_1(x_1, \ldots, x_n) \\ x_2 &= f_2(x_1, \ldots, x_n) \\ &\vdots \\ x_n &= f_n(x_1, \ldots, x_n) \end{aligned}$$

where x_i are variables and $f_i : L^n \to L$ is a collection of functions. A *solution* to an equation system provides a value from L for each variable such that all equations are satisfied.

We can combine the n functions into one, $f : L^n \to L^n$,

$$f(x_1, \ldots, x_n) = \big(f_1(x_1, \ldots, x_n), \ldots, f_n(x_1, \ldots, x_n)\big)$$

in which case the equation system looks like

$$x = f(x)$$

where $x \in L^n$. This clearly shows that a solution to an equation system is the same as a fixed-point of its functions. As we aim for the most precise solutions, we want *least* fixed-points.

Exercise 4.23: Show that f is monotone if and only if each $f_1, \ldots, f_n$ is monotone, where f is defined from $f_1, \ldots, f_n$ as above.

As an example, for the equation system from earlier in this section

$x_1 = [\mathsf{a} \mapsto ?, \mathsf{b} \mapsto ?]$
$x_2 = x_1[\mathsf{a} \mapsto +]$
$x_3 = x_2[\mathsf{b} \mapsto x_2(\mathsf{a}) + ?]$
$x_4 = x_3[\mathsf{a} \mapsto x_3(\mathsf{a}) - x_3(\mathsf{b})]$

we have four constraint variables, $x_1, \ldots, x_4$ with constraint functions $f_1, \ldots, f_4$ defined as follows:

$f_1(x_1, \ldots, x_4) = [\mathsf{a} \mapsto ?, \mathsf{b} \mapsto ?]$
$f_2(x_1, \ldots, x_4) = x_1[\mathsf{a} \mapsto +]$
$f_3(x_1, \ldots, x_4) = x_2[\mathsf{b} \mapsto x_2(\mathsf{a}) + ?]$
$f_4(x_1, \ldots, x_4) = x_3[\mathsf{a} \mapsto x_3(\mathsf{a}) - x_3(\mathsf{b})]$

Exercise 4.24: Show that the four constraint functions $f_1, \ldots, f_4$ are monotone. (Hint: see exercise 4.22.)

As mentioned earlier, for this simple equation system it is trivial to find a solution by substitution, however, that method is inadequate for equation systems that arise when analyzing programs more generally.

Exercise 4.25: Argue that your solution from Exercise 4.12 is the least fixed-point of the function f defined by
$f(x_1, \ldots, x_4) = \big(f_1(x_1, \ldots, x_4), \ldots, f_4(x_1, \ldots, x_4)\big)$.

The central result we need is the *fixed-point theorem*:[6]

In a lattice L with finite height, every monotone function $f : L \to L$ has a unique least fixed-point denoted $\mathit{fix}(f)$ defined as:

$$\mathit{fix}(f) = \bigsqcup_{i \geq 0} f^i(\bot)$$

The proof of this theorem is quite simple. Observe that $\bot \sqsubseteq f(\bot)$ since $\bot$ is the least element. Since f is monotone, it follows that $f(\bot) \sqsubseteq f^2(\bot)$ and by induction that $f^i(\bot) \sqsubseteq f^{i+1}(\bot)$ for any i. Thus, we have an increasing chain:

$$\bot \sqsubseteq f(\bot) \sqsubseteq f^2(\bot) \sqsubseteq \ldots$$

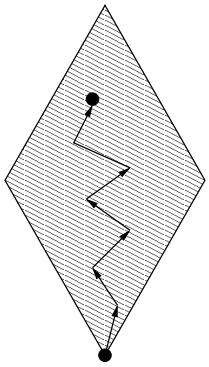
Since L is assumed to have finite height, we must for some k have that $f^k(\bot) = f^{k+1}(\bot)$, i.e. $f^k(\bot)$ is a fixed-point for f. By Exercise 4.2, $f^k(\bot)$ must be the least upper bound of all elements in the chain, so $\mathit{fix}(f) = f^k(\bot)$. Assume now that x is another fixed-point. Since $\bot \sqsubseteq x$ it follows that $f(\bot) \sqsubseteq f(x) = x$, since f is monotone, and by induction we get that $\mathit{fix}(f) = f^k(\bot) \sqsubseteq x$. Hence, $\mathit{fix}(f)$ is a least fixed-point, and by anti-symmetry of $\sqsubseteq$ it is also unique.

[6] There are many fixed-point theorems in the literature; the one we use here is a variant of one by Kleene.

The theorem is a powerful result: It tells us not only that equation systems over lattices always have solutions, provided that the lattices have finite height and the constraint functions are monotone, but also that uniquely most precise solutions always exist. Furthermore, the careful reader may have noticed that the theorem provides an algorithm for computing the least fixed-point: simply compute the increasing chain $\bot \sqsubseteq f(\bot) \sqsubseteq f^2(\bot) \sqsubseteq \ldots$ until the fixed-point is reached. In pseudo-code, this so-called *naive fixed-point algorithm* looks as follows.

```
procedure NaiveFixedPointAlgorithm(f)
    x := ⊥
    while x ≠ f(x) do
        x := f(x)
    end while
    return x
end procedure
```

(Instead of computing $f(x)$ both in the loop condition and in the loop body, a trivial improvement is to just compute it once in each iteration and see if the result changes.) The computation of a fixed-point can be illustrated as a walk up the lattice starting at $\bot$:



This algorithm is called "naive" because it does not exploit the special structures that are common in analysis lattices. We shall see various less naive fixed-point algorithms in Section 5.3.

The least fixed point is the most precise possible solution to the equation system, but the equation system is (for a sound analysis) merely a conservative approximation of the actual program behavior (again, recall the `TM(j)` example from Chapter 1). This means that the semantically most precise possible (while still correct) answer is generally *below* the least fixed point in the lattice. We shall see examples of this in Chapter 5.

Exercise 4.26: Explain step-by-step how the naive fixed-point algorithm computes the solution to the equation system from Exercise 4.12.

The time complexity of computing a fixed-point with this algorithm depends on

- the height of the lattice, since this provides a bound for the number of iterations of the algorithm, and
- the cost of computing $f(x)$ and testing equality, which are performed in each iteration.

We shall investigate other properties of this algorithm and more sophisticated variants in Section 5.3.

Exercise 4.27: Does the fixed-point theorem also hold without the assumption that the lattice has finite height? If yes, give a proof; if no, give a counter-example.

We can similarly solve systems of *inequations* of the form

$$\begin{aligned} x_1 &\sqsupseteq f_1(x_1, \ldots, x_n) \\ x_2 &\sqsupseteq f_2(x_1, \ldots, x_n) \\ &\vdots \\ x_n &\sqsupseteq f_n(x_1, \ldots, x_n) \end{aligned}$$

by observing that the relation $x \sqsupseteq y$ is equivalent to $x = x \sqcup y$ (see Exercise 4.2). Thus, solutions are preserved by rewriting the system into

$$\begin{aligned} x_1 &= x_1 \sqcup f_1(x_1, \ldots, x_n) \\ x_2 &= x_2 \sqcup f_2(x_1, \ldots, x_n) \\ &\vdots \\ x_n &= x_n \sqcup f_n(x_1, \ldots, x_n) \end{aligned}$$

which is just a system of equations with monotone functions as before (see Exercises 4.18 and 4.19). Conversely, constraints of the form

$$\begin{aligned} x_1 &\sqsubseteq f_1(x_1, \ldots, x_n) \\ x_2 &\sqsubseteq f_2(x_1, \ldots, x_n) \\ &\vdots \\ x_n &\sqsubseteq f_n(x_1, \ldots, x_n) \end{aligned}$$

can be rewritten into

$$\begin{aligned} x_1 &= x_1 \sqcap f_1(x_1, \ldots, x_n) \\ x_2 &= x_2 \sqcap f_2(x_1, \ldots, x_n) \\ &\vdots \\ x_n &= x_n \sqcap f_n(x_1, \ldots, x_n) \end{aligned}$$

by observing that the relation $x \sqsubseteq y$ is equivalent to $x = x \sqcap y$.

Chapter 5

Dataflow Analysis with Monotone Frameworks

Classical dataflow analysis starts with a CFG and a lattice with finite height. The lattice describes abstract information we wish to infer for the different CFG nodes. It may be fixed for all programs, or it may be parameterized based on the given program. To every node v in the CFG, we assign a constraint variable[1] $[\![v]\!]$ ranging over the elements of the lattice. For each node we then define a *dataflow constraint* that relates the value of the variable of the node to those of other nodes (typically the neighbors), depending on what construction in the programming language the node represents. If all the constraints for the given program happen to be equations or inequations with monotone right-hand sides, then we can use the fixed-point algorithm from Section 4.4 to compute the analysis result as the unique least solution.

The combination of a lattice and a space of monotone functions is called a *monotone framework*. For a given program to be analyzed, a monotone framework can be instantiated by specifying the CFG and the rules for assigning dataflow constraints to its nodes.

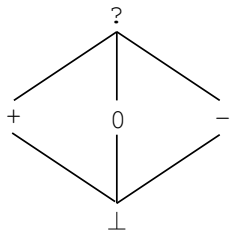
An analysis is *sound* if all solutions to the constraints correspond to correct information about the program. The solutions may be more or less imprecise, but computing the least solution will give the highest degree of precision possible.

Throughout this chapter we use the subset of TIP without function calls and pointers; those language features are studied in Chapters 8 and 9.

[1] As for type analysis, we will ambiguously use the notation $[\![S]\!]$ for $[\![v]\!]$ if S is the syntax associated with v. The meaning will always be clear from the context.

5.1 Sign Analysis, Revisited

Continuing the example from Section 4.1, our goal is to determine the sign (positive, zero, negative) of all expressions in the given programs. We start with the tiny lattice $Sign$ for describing abstract values:



We want an abstract value for each program variable, so we define the map lattice

$$States = Vars \to Sign$$

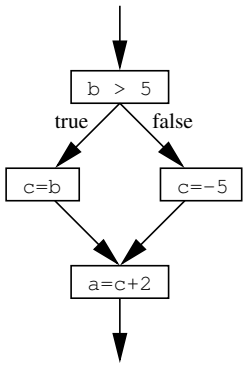
where $Vars$ is the set of variables occurring in the given program. Each element of this lattice can be thought of as an abstract state, hence its name. For each CFG node v we assign a constraint variable $[\![v]\!]$ denoting an abstract state that gives the sign values for all variables at the program point immediately after v. The lattice $States^n$, where n is the number of CFG nodes, then models information for all the CFG nodes.

The dataflow constraints model the effects of program execution on the abstract states. For simplicity, we here focus on a subset of TIP that does not contain pointers or function calls, so integers are the only type of values we need to consider.

First, we define an auxiliary function $JOIN(v)$ that combines the abstract states from the predecessors of a node v:

$$JOIN(v) = \bigsqcup_{w \in pred(v)} [\![w]\!]$$

For example, with the following CFG, we have $JOIN([\![\texttt{a=c+2}]\!]) = [\![\texttt{c=b}]\!] \sqcup [\![\texttt{c=-5}]\!]$.



The most interesting constraint rule for this analysis is the one for assignment statements, that is, nodes v of the form $X = E$:

$X = E$: $\quad [\![v]\!] = JOIN(v)[X \mapsto eval(JOIN(v), E)]$

This constraint rule models the fact that the abstract state after an assignment $X = E$ is equal to the abstract state immediately before the assignment, except that the abstract value of X is the result of abstractly evaluating the expression E. The $eval$ function performs an abstract evaluation of expression E relative to an abstract state σ:

$eval(\sigma, X) = \sigma(X)$
$eval(\sigma, I) = sign(I)$
$eval(\sigma, E_1 \,\mathtt{op}\, E_2) = \widehat{\mathtt{op}}(eval(\sigma, E_1), eval(\sigma, E_2))$

The function $sign$ gives the sign of an integer constant, and $\widehat{\mathtt{op}}$ is an abstract evaluation of the given operator,[2] defined by the following tables:

$\widehat{+}$	$\bot$	0	-	+	?
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
0	$\bot$	0	-	+	?
-	$\bot$	-	-	?	?
+	$\bot$	+	?	+	?
?	$\bot$	?	?	?	?

$\widehat{-}$	$\bot$	0	-	+	?
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
0	$\bot$	0	+	-	?
-	$\bot$	-	?	-	?
+	$\bot$	+	+	?	?
?	$\bot$	?	?	?	?

$\widehat{*}$	$\bot$	0	-	+	?
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
0	$\bot$	0	0	0	0
-	$\bot$	0	+	-	?
+	$\bot$	0	-	+	?
?	$\bot$	0	?	?	?

$\widehat{/}$	$\bot$	0	-	+	?
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
0	$\bot$	$\bot$	0	0	?
-	$\bot$	$\bot$	?	?	?
+	$\bot$	$\bot$	?	?	?
?	$\bot$	$\bot$	?	?	?

$\widehat{>}$	$\bot$	0	-	+	?
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
0	$\bot$	0	+	0	?
-	$\bot$	0	?	0	?
+	$\bot$	+	+	?	?
?	$\bot$	?	?	?	?

$\widehat{==}$	$\bot$	0	-	+	?
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
0	$\bot$	+	0	0	?
-	$\bot$	0	?	0	?
+	$\bot$	0	0	?	?
?	$\bot$	?	?	?	?

Variable declarations are modeled as follows (recall that freshly declared local variables are uninitialized, so they can have any value).

$\mathtt{var}\ X_1, \ldots, X_n$: $\quad [\![v]\!] = JOIN(v)[X_1 \mapsto \mathtt{?}, \ldots, X_n \mapsto \mathtt{?}]$

For the subset of TIP we have chosen to focus on, no other kinds of CFG nodes affect the values of variables, so for the remaining nodes we have this trivial constraint rule:

$$[\![v]\!] = JOIN(v)$$

[2] Unlike in Section 4.4, to avoid confusion we now distinguish between concrete operators and their abstract counterparts using the $\widehat{\cdots}$ notation.

Exercise 5.1: In the CFGs we consider in this chapter (for TIP without function calls), entry nodes have no predecessors.

(a) Argue that the constraint rule $[\![v]\!] = JOIN(v)$ for such nodes is equivalent to defining $[\![v]\!] = \bot$.

(b) Argue that removing all equations of the form $[\![v]\!] = \bot$ from an equation system does not change its least solution.

The lattice and constraints form a monotone framework. To see that all the right-hand sides of our constraints correspond to monotone functions, notice that they are all composed (see Exercise 4.18) from the $\sqcup$ operator (see Exercise 4.19), map updates (see Exercise 4.22), and the $eval$ function. The $sign$ function is constant (see Exercise 4.15). Monotonicity of the abstract operators used by $eval$ can be verified by a tedious manual inspection. For a lattice with n elements, monotonicity of an $n \times n$ table can be verified automatically in time $\mathcal{O}(n^3)$.

Exercise 5.2: Describe an algorithm for checking monotonicity of an operator given by an $n \times n$ table. Can you do better than $\mathcal{O}(n^3)$ time?

Exercise 5.3: Check that the above tables indeed define monotone operators on the $Sign$ lattice.

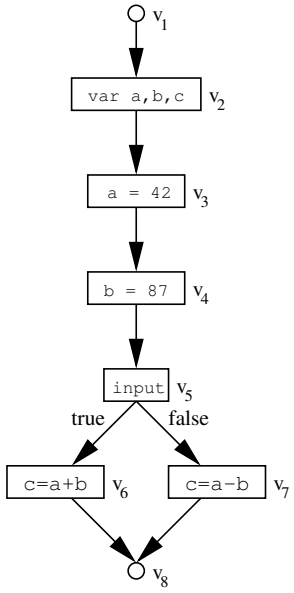
Exercise 5.4: Argue that these tables are the most precise possible for the $Sign$ lattice, given that soundness must be preserved.

Exercise 5.5: The table for the abstract evaluation of == is unsound if we consider the full TIP language instead of the subset without pointers or function calls. Why? And how could it be fixed?

Recall the example program from Section 4.1:

```
var a,b,c;
a = 42;
b = 87;
if (input) {
  c = a + b;
} else {
  c = a - b;
}
```

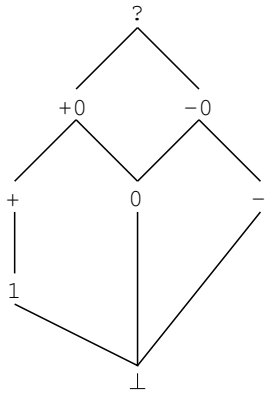
Its CFG looks as follows, with nodes $\{v_1, \ldots, v_8\}$:



Exercise 5.6: Generate the equation system for this example program. Then solve it using the fixed-point algorithm from Section 4.4.

Exercise 5.7: Write a small TIP program where the sign analysis leads to an equation system with mutually recursive constraints. Then explain step-by-step how the fixed-point algorithm from Section 4.4 computes the solution.

We lose some information in the above analysis, since for example the expressions `(2>0)==1` and `x-x` are analyzed as `?`, which seems unnecessarily coarse. (These are examples where the least fixed-point of the analysis equation system is not identical to the semantically best possible answer.) Also, the expression `+/+` results in `?` rather than `+` since e.g. `1/2` is rounded down to zero. To handle some of these situations more precisely, we could enrich the sign lattice with element `1` (the constant 1), `+0` (positive or zero), and `-0` (negative or zero) to keep track of more precise abstract values:



and consequently describe the abstract operators by 8×8 tables.

Exercise 5.8: Define the six operators on the extended $Sign$ lattice by means of 8×8 tables. Check that they are monotone.

Exercise 5.9: Show how the $eval$ function could be improved to make the sign analysis able to show that the final value of `z` cannot be a negative number in the following program:

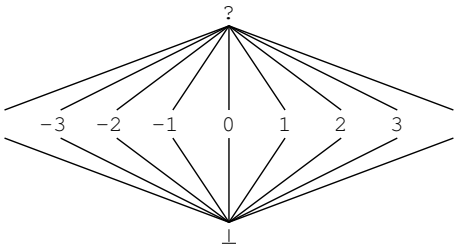
```
var x,y,z;
x = input;
y = x*x;
z = (x-x+1)*y;
```

The results of a sign analysis could in theory be used to eliminate division-by-zero errors by rejecting programs in which denominator expressions have sign `0` or `?`. However, the resulting analysis will probably unfairly reject too many programs to be practical. Other more powerful analysis techniques, such as interval analysis (Section 5.11) and path sensitivity (Chapter 6) would be more useful for detecting such errors.

5.2 Constant Propagation Analysis

An analysis related to sign analysis is constant propagation analysis, where we for every program point want to determine the variables that have a constant value. The analysis is structured just like the sign analysis, except for two modifications. First, the $Sign$ lattice is replaced by $flat(\mathbb{Z})$ where $\mathbb{Z}$ is the set of all integers:[3]

[3]For simplicity, assume that TIP integer values are unbounded.



Second, the abstraction of operators $\mathtt{op} \in \{+, -, *, /, >, ==\}$ is modified accordingly:

$$a \mathbin{\widehat{\mathtt{op}}} b = \begin{cases} \bot & \text{if } a = \bot \text{ or } b = \bot \\ ? & \text{if } a = ? \text{ or } b = ? \\ a \mathbin{\mathtt{op}} b & \text{if } a, b \in \mathbb{Z} \end{cases}$$

Exercise 5.10: Argue that this definition of $\widehat{\mathtt{op}}$ leads to a sound analysis.

Using constant propagation analysis, an optimizing compiler could transform the program

```
var x,y,z;
x = 27;
y = input;
z = 2*x+y;
if (x < 0) { y = z-3; } else { y = 12; }
output y;
```

into

```
var x,y,z;
x = 27;
y = input;
z = 54+y;
if (0) { y = z-3; } else { y = 12; }
output y;
```

which, following a reaching definitions analysis and dead code elimination (see Section 5.7), can be reduced to this shorter and more efficient program:

```
var y;
y = input;
output 12;
```

Exercise 5.11: Assume that TIP computes with (arbitrary-precision) real numbers instead of integers. Design an analysis that finds out which variables at each program point in a given program only have integer values.

5.3 Fixed-Point Algorithms

In summary, dataflow analysis works as follows. For a CFG with nodes $Nodes = \{v_1, v_2, \ldots, v_n\}$ we work in the lattice L^n where L is a lattice that models abstract states. Assuming that node v_i generates the dataflow equation $[\![v_i]\!] = f_i([\![v_1]\!], \ldots, [\![v_n]\!])$, we construct the combined function $f : L^n \to L^n$ by defining $f(x_1, \ldots, x_n) = \big(f_1(x_1, \ldots, x_n), \ldots, f_n(x_1, \ldots, x_n)\big)$. Applying the fixed-point algorithm, NaiveFixedPointAlgorithm(f) (see page 37), then gives us the desired solution for $[\![v_1]\!], \ldots, [\![v_n]\!]$.

Exercise 4.26 (page 37) demonstrates why the algorithm is called "naive". In each iteration it applies all the constraint functions, $f_1, \ldots, f_4$, and much of that computation is redundant. For example, f_2 (see page 36) depends only on x_1, but the value of x_1 is unchanged in most iterations.

As a step toward more efficient algorithms, the *round-robin algorithm* exploits the fact that our lattice has the structure L^n and that f is composed from $f_1, \ldots, f_n$:

```
procedure RoundRobin(f_1, ..., f_n)
    (x_1, ..., x_n) := (⊥, ..., ⊥)
    while (x_1, ..., x_n) ≠ f(x_1, ..., x_n) do
        for i := 1 ... n do
            x_i := f_i(x_1, ..., x_n)
        end for
    end while
    return (x_1, ..., x_n)
end procedure
```

(Similar to the naive fixed-point algorithm, it is trivial to avoid computing each $f_i(x_1, \ldots, x_n)$ twice in every iteration.) Notice that one iteration of the while-loop in this algorithm does not in general give the same result as one iteration of the naive fixed-point algorithm: when computing $f_i(x_1, \ldots, x_n)$, the values of $x_1, \ldots, x_{i-1}$ have been updated by the preceding iterations of the inner loop (while the values of $x_i, \ldots, x_n$ come from the previous iteration of the outer loop or are still $\bot$, like in the naive fixed-point algorithm). Nevertheless, the algorithm always terminates and produces the same result as the naive fixed-point algorithm. Each iteration of the while-loop takes the same time as for the naive fixed-point algorithm, but the number of iterations required to reach the fixed-point may be lower.

Exercise 5.12: Prove that the round-robin algorithm computes the least fixed-point of f. (Hint: see the proof of the fixed-point theorem, and consider the ascending chain that arises from the sequence of $x_i := f_i(x_1, \ldots, x_n)$ operations.)

Exercise 5.13: Continuing Exercise 4.26, how many iterations are required by the naive fixed-point algorithm and the round-robin algorithm, respectively, to reach the fixed-point?

We can do better than round-robin. First, the order of the iterations $i := 1 \ldots n$ is clearly irrelevant for the correctness of the algorithm (see your proof from Exercise 5.12). Second, we still apply all constraint functions in each iteration of the repeat-until loop. What matters for correctness is, which should be clear from your solution to Exercise 5.12, that the constraint functions are applied until the fixed-point is reached for all of them. This observation leads to the *chaotic-iteration algorithm*:

procedure ChaoticIteration($f_1, \ldots, f_n$)
  $(x_1, \ldots, x_n) := (\bot, \ldots, \bot)$
  while $(x_1, \ldots, x_n) \neq f(x_1, \ldots, x_n)$ **do**
    choose i **nondeterministically from** $\{1, \ldots, n\}$
    $x_i := f_i(x_1, \ldots, x_n)$
  end while
  return $(x_1, \ldots, x_n)$
end procedure

This is not a practical algorithm, because its efficiency and termination depend on how i is chosen in each iteration. Additionally, computing the loop condition is now more expensive than executing the loop body. However, *if* it terminates, the algorithm produces the right result.

Exercise 5.14: Prove that the chaotic-iteration algorithm computes the least fixed-point of f, if it terminates. (Hint: see your solution to Exercise 5.12.)

The algorithm we describe next is a practical variant of chaotic-iteration.

In the general case, every constraint variable $[\![v_i]\!]$ may depend on all other variables. Most often, however, an actual instance of f_i will only read the values of a few other variables, as in the examples from Exercise 4.24 and Exercise 5.6. We represent this information as a map

$$dep : Nodes \to 2^{Nodes}$$

which for each node v tells us the subset of other nodes for which $[\![v]\!]$ occurs in a nontrivial manner on the right-hand side of their dataflow equations. That is, $dep(v)$ is the set of nodes whose information may depend on the information of v. We also define its inverse: $dep^{-1}(v) = \{w \mid v \in dep(w)\}$.

For the example from Exercise 5.6, we have, for instance, $dep(v_5) = \{v_6, v_7\}$. This means that whenever $[\![v_5]\!]$ changes its value during the fixed-point computation, only f_6 and f_7 need to be recomputed.

Armed with this information, we can present a simple *work-list algorithm*:

```
procedure SimpleWorkListAlgorithm(f_1, ..., f_n)
    (x_1, ..., x_n) := (⊥, ..., ⊥)
    W := {v_1, ..., v_n}
    while W ≠ ∅ do
        v_i := W.removeNext()
        y := f_i(x_1, ..., x_n)
        if y ≠ x_i then
            x_i := y
            for each v_j ∈ dep(v_i) do
                W.add(v_j)
            end for
        end if
    end while
    return (x_1, ..., x_n)
end procedure
```

The set W is here called the work-list with operations 'add' and 'removeNext' for adding and (nondeterministically) removing an item. The work-list initially contains all nodes, so each f_i is applied at least once. It is easy to see that the work-list algorithm terminates on any input: In each iteration, we either move up in the L^n lattice, or the size of the work-list decreases. As usual, we can only move up in the lattice finitely many times as it has finite height, and the while-loop terminates when the work-list is empty. Correctness follows from observing that each iteration of the algorithm has the same effect on $(x_1, \ldots, x_n)$ as one iteration of the chaotic-iteration algorithm for some nondeterministic choice of i.

Exercise 5.15: Argue that a sound, but probably not very useful choice for the dep map is one that always returns the set of all CFG nodes.

Exercise 5.16: As stated above, we can choose $dep(v_5) = \{v_6, v_7\}$ for the example equation system from Exercise 5.6. Argue that a good strategy for the sign analysis is to define $dep = succ$. (We return to this topic in Section 5.8.)

Exercise 5.17: Explain step-by-step how the work-list algorithm computes the solution to the equation system from Exercise 5.6. (Since the 'removeNext' operation is nondeterministic, there are many solutions!)

Exercise 5.18: When reasoning about worst-case complexity of analyses that are based on work-list algorithms, it is sometimes useful if one can bound the number of predecessors $|pred(v)|$ or successors $|succ(v)|$ for all nodes v.

(a) Describe a family of TIP functions where the maximum number of successors $|succ(v)|$ for the nodes v in each function grows linearly in the number of CFG nodes.

(b) Now let us modify the CFG construction slightly, such that a dummy "no-op" node is inserted at the merge point after the two branches of each `if` block. This will increase the number of CFG nodes by at most a constant factor. Argue that we now have $|pred(v)| \leq 2$ and $|succ(v)| \leq 2$ for all nodes v.

Assuming that $|dep(v)|$ and $|dep^{-1}(v)|$ are bounded by a constant for all nodes v, the worst-case time complexity of the simple work-list algorithm can be expressed as

$$\mathcal{O}(n \cdot h \cdot k)$$

where n is the number of CFG nodes in the program being analyzed, h is the height of the lattice L for abstract states, and k is the worst-case time required to compute a constraint function $f_i(x_1, \ldots, x_n)$.

Exercise 5.19: Prove the above statement about the worst-case time complexity of the simple work-list algorithm. (It is reasonable to assume that the work-list operations 'add' and 'removeNext' take constant time.)

Exercise 5.20: Another useful observation when reasoning about worst-case complexity of dataflow analyses is that normalizing a program (see Section 2.3) may increase the number of CFG nodes by more than a constant factor, but represented as an AST or as textual source code, the size of the program increases by at most a constant factor. Explain why this claim is correct.

Exercise 5.21: Estimate the worst-case time complexity of the sign analysis with the simple work-list algorithm, using the formula above. (As this formula applies to *any* dataflow analysis implemented with the simple work-list algorithm, the actual worst-case complexity of this specific analysis may be asymptotically better!)

Further algorithmic improvements are possible. It may be beneficial to handle in separate turns the strongly connected components of the graph induced by the dep map, and the worklist set could be changed into a priority queue allowing us to exploit domain-specific knowledge about a particular dataflow problem. Also, for some analyses, the dependence information can be made more precise

by allowing dep to consider the current value of $(x_1, \ldots, x_n)$ in addition to the node v.

5.4 Live Variables Analysis

A variable is *live* at a program point if there exists an execution where its value is read later in the execution without it is being written to in between. Clearly undecidable, this property can be approximated by a static analysis called live variables analysis (or liveness analysis). The typical use of live variables analysis is optimization: there is no need to store the value of a variable that is not live. For this reason, we want the analysis to be conservative in the direction where the answer "not live" can be trusted and "live" is the safe but useless answer.

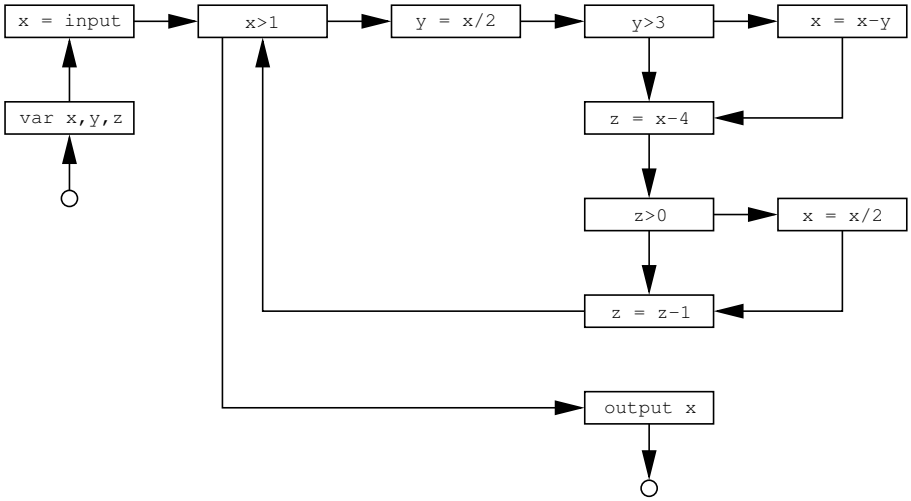
We use a powerset lattice where the elements are the variables occurring in the given program. This is an example of a *parameterized* lattice, that is, one that depends on the specific program being analyzed. For the example program

```
var x,y,z;
x = input;
while (x>1) {
  y = x/2;
  if (y>3) x = x-y;
  z = x-4;
  if (z>0) x = x/2;
  z = z-1;
}
output x;
```

the lattice modeling abstract states is thus:

$$States = (2^{\{\texttt{x,y,z}\}}, \subseteq)$$

The corresponding CFG looks as follows:



For every CFG node v we introduce a constraint variable $[\![v]\!]$ denoting the subset of program variables that are live at the program point *before* that node. The analysis wil be conservative, since the computed set may be too large. We use the auxiliary definition

$$JOIN(v) = \bigcup_{w \in succ(v)} [\![w]\!]$$

Unlike the $JOIN$ function from sign analysis, this one combines abstract states from the successors instead of the predecessors. We have defined the order relation as $\sqsubseteq = \subseteq$, so $\sqcup = \cup$.

As in sign analysis, the most interesting constraint rule is the one for assignments:

X = E: $\quad [\![v]\!] = JOIN(v) \setminus \{X\} \cup vars(E)$

This rule models the fact that the set of live variables before the assignment is the same as the set after the assignment, except for the variable being written to and the variables that are needed to evaluate the right-hand-side expression.

Exercise 5.22: Explain why the constraint rule for assignments, as defined above, is sound.

Branch conditions and output statements are modelled as follows:

`if (`E`):`
`while (`E`):`
`output` E`:` $\quad [\![v]\!] = JOIN(v) \cup vars(E)$

where $vars(E)$ denotes the set of variables occurring in E. For variable declarations and exit nodes:

`var` X_1`,` $\ldots$`,` X_n`:` $\quad [\![v]\!] = JOIN(v) \setminus \{X_1, \ldots, X_n\}$

$$[\![\mathit{exit}]\!] = \emptyset$$

For all other nodes:

$$[\![v]\!] = \mathit{JOIN}(v)$$

Exercise 5.23: Argue that the right-hand sides of the constraints define monotone functions.

Our example program yields these constraints:

$[\![\texttt{var x,y,z}]\!] = [\![\texttt{x=input}]\!] \setminus \{\texttt{x},\texttt{y},\texttt{z}\}$
$[\![\texttt{x=input}]\!] = [\![\texttt{x>1}]\!] \setminus \{\texttt{x}\}$
$[\![\texttt{x>1}]\!] = ([\![\texttt{y=x/2}]\!] \cup [\![\texttt{output x}]\!]) \cup \{\texttt{x}\}$
$[\![\texttt{y=x/2}]\!] = ([\![\texttt{y>3}]\!] \setminus \{\texttt{y}\}) \cup \{\texttt{x}\}$
$[\![\texttt{y>3}]\!] = [\![\texttt{x=x-y}]\!] \cup [\![\texttt{z=x-4}]\!] \cup \{\texttt{y}\}$
$[\![\texttt{x=x-y}]\!] = ([\![\texttt{z=x-4}]\!] \setminus \{\texttt{x}\}) \cup \{\texttt{x},\texttt{y}\}$
$[\![\texttt{z=x-4}]\!] = ([\![\texttt{z>0}]\!] \setminus \{\texttt{z}\}) \cup \{\texttt{x}\}$
$[\![\texttt{z>0}]\!] = [\![\texttt{x=x/2}]\!] \cup [\![\texttt{z=z-1}]\!] \cup \{\texttt{z}\}$
$[\![\texttt{x=x/2}]\!] = ([\![\texttt{z=z-1}]\!] \setminus \{\texttt{x}\}) \cup \{\texttt{x}\}$
$[\![\texttt{z=z-1}]\!] = ([\![\texttt{x>1}]\!] \setminus \{\texttt{z}\}) \cup \{\texttt{z}\}$
$[\![\texttt{output x}]\!] = [\![\mathit{exit}]\!] \cup \{\texttt{x}\}$
$[\![\mathit{exit}]\!] = \emptyset$

whose least solution is:

$[\![\mathit{entry}]\!] = \emptyset$
$[\![\texttt{var x,y,z}]\!] = \emptyset$
$[\![\texttt{x=input}]\!] = \emptyset$
$[\![\texttt{x>1}]\!] = \{\texttt{x}\}$
$[\![\texttt{y=x/2}]\!] = \{\texttt{x}\}$
$[\![\texttt{y>3}]\!] = \{\texttt{x},\texttt{y}\}$
$[\![\texttt{x=x-y}]\!] = \{\texttt{x},\texttt{y}\}$
$[\![\texttt{z=x-4}]\!] = \{\texttt{x}\}$
$[\![\texttt{z>0}]\!] = \{\texttt{x},\texttt{z}\}$
$[\![\texttt{x=x/2}]\!] = \{\texttt{x},\texttt{z}\}$
$[\![\texttt{z=z-1}]\!] = \{\texttt{x},\texttt{z}\}$
$[\![\texttt{output x}]\!] = \{\texttt{x}\}$
$[\![\mathit{exit}]\!] = \emptyset$

From this information a clever compiler could deduce that y and z are never live at the same time, and that the value written in the assignment z=z-1 is never read. Thus, the program may safely be optimized into the following one, which saves the cost of one assignment and could result in better register allocation:

```
var x,yz;
x = input;
while (x>1) {
```

```
    yz = x/2;
    if (yz>3) x = x-yz;
    yz = x-4;
    if (yz>0) x = x/2;
  }
  output x;
```

Exercise 5.24: Consider the following program:

```
main() {
  var x,y,z;
  x = input;
  y = input;
  z = x;
  output y;
}
```

Show for each program point the set of live variables, as computed by our live variables analysis. (Do not forget the entry and exit points.)

Exercise 5.25: An analysis is distributive if all its constraint functions are distributive according to the definition from Exercise 4.16. Show that live variables analysis is distributive.

Exercise 5.26: As Exercise 5.24 demonstrates, live variables analysis is not ideal for locating code that can safely be removed, if building an optimizing compiler. Let us define that a variable is *useless* at a given program point if it is dead (i.e. not live) or its value is only used to compute values of useless variables. A variable is *strongly live* if it is not useless.

(a) Show how the live variables analysis can be modified to compute strongly live variables.

(b) Show for each program point in the program from Exercise 5.24 the set of strongly live variables, as computed by your new analysis.

We can estimate the worst-case time complexity of the live variables analysis, with for example the naive fixed-point algorithm from Section 4.4. We first observe that if the program has n CFG nodes and b variables, then the lattice $(2^{Vars})^n$ has height $b \cdot n$, which bounds the number of iterations we can perform. Each lattice element can be represented as a bitvector of length $b \cdot n$. Using the observation from Exercise 5.18 we can ensure that $|succ(v)| < 2$ for any node v. For each iteration we therefore have to perform $\mathcal{O}(n)$ intersection, difference, or equality operations on sets of size b, which can be done in time $\mathcal{O}(b \cdot n)$. Thus, we reach a time complexity of $\mathcal{O}(b^2 \cdot n^2)$.

Exercise 5.27: Can you obtain an asymptotically better bound on the worst-case time complexity of live variables analysis with the naive fixed-point algorithm, if exploiting properties of the structures of TIP CFGs and the analysis constraints?

Exercise 5.28: Recall from Section 5.3 that the work-list algorithm relies on a function $dep(v)$ for avoiding recomputation of constraint functions that are guaranteed not to change outputs. What would be a good strategy for defining $dep(v)$ in general for live variables analysis of any given program?

Exercise 5.29: Estimate the worst-case time complexity of the live variables analysis with the simple work-list algorithm, by using the formula from page 49.

5.5 Available Expressions Analysis

A nontrivial expression in a program is *available* at a program point if its current value has already been computed earlier in the execution. Such information is useful for program optimization. The set of available expressions for all program points can be approximated using a dataflow analysis. The lattice we use has as elements all expressions occurring in the program. To be useful for program optimization purposes, an expression may be included at a given program point only if it is *definitely* available not matter how the computation arrived at that program point, so we choose the lattice to be ordered by *reverse* subset inclusion.

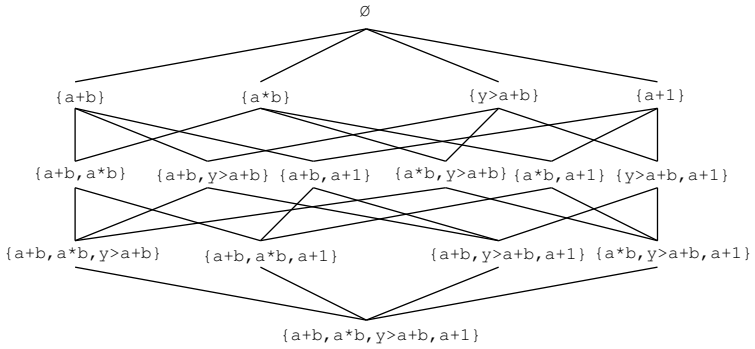
For the program

```
var x,y,z,a,b;
z = a+b;
y = a*b;
while (y > a+b) {
  a = a+1;
  x = a+b;
}
```

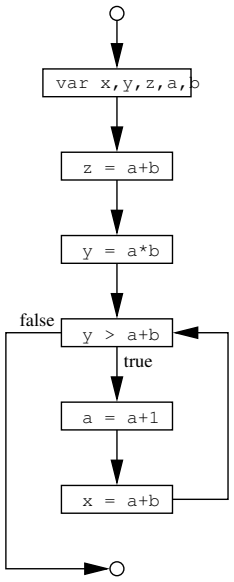
we have four different nontrivial expressions, so our lattice for abstract states is

$$States = (2^{\{\texttt{a+b},\texttt{a*b},\texttt{y>a+b},\texttt{a+1}\}}, \supseteq)$$

which looks like this:



The top element of our lattice is $\emptyset$, which corresponds to the trivial information that no expressions are known to be available. The CFG for above program looks as follows:



As usual in dataflow analysis, for each CFG node v we introduce a constraint variable $[\![v]\!]$ ranging over $States$. Our intention is that it should contain the subset of expressions that are guaranteed always to be available at the program point after that node. For example, the expression `a+b` is available at the condition in the loop, but it is not available at the final assignment in the loop. Our analysis will be conservative in the sense that the computed sets may be too small but never too large.

Next we define the dataflow constraints. The intuition is that an expression is available at a node v if it is available from all incoming edges or is computed by v, unless its value is destroyed by an assignment statement.

The $JOIN$ function uses $\cap$ (because the lattice order is now $\supseteq$) and $pred$

(because availability of expressions depends on information from the past):

$$JOIN(v) = \bigcap_{w \in pred(v)} [\![w]\!]$$

Assignments are modeled as follows:

X = E: $\quad [\![v]\!] = (JOIN(v) \cup exps(E)) \downarrow X$

Here, the function $\downarrow X$ removes all expressions that contain the variable X, and $exps$ collects all nontrivial expressions:

$exps(I) = \emptyset$
$exps(X) = \emptyset$
$exps(\texttt{input}) = \emptyset$
$exps(E_1 \texttt{ op } E_2) = \{E_1 \texttt{ op } E_2\} \cup exps(E_1) \cup exps(E_2)$

No expressions are available at entry nodes:

$$[\![entry]\!] = \emptyset$$

Branch conditions and output statements accumulate more available expressions:

$$\left.\begin{array}{l}\texttt{if (}E\texttt{):}\\ \texttt{while (}E\texttt{):}\\ \texttt{output } E\texttt{:}\end{array}\right\} \quad [\![v]\!] = JOIN(v) \cup exps(E)$$

For all other kinds of nodes, the collected sets of expressions are simply propagated from the predecessors:

$$[\![v]\!] = JOIN(v)$$

Again, the right-hand sides of all constraints are monotone functions.

Exercise 5.30: Explain informally why the constraints are monotone and the analysis is sound.

For the example program, we generate the following constraints:

$[\![entry]\!] = \emptyset$
$[\![\texttt{var x,y,z,a,b}]\!] = [\![entry]\!]$
$[\![\texttt{z=a+b}]\!] = exps(\texttt{a+b}) \downarrow \texttt{z}$
$[\![\texttt{y=a*b}]\!] = ([\![\texttt{z=a+b}]\!] \cup exps(\texttt{a*b})) \downarrow \texttt{y}$
$[\![\texttt{y>a+b}]\!] = ([\![\texttt{y=a*b}]\!] \cap [\![\texttt{x=a+b}]\!]) \cup exps(\texttt{y>a+b})$
$[\![\texttt{a=a+1}]\!] = ([\![\texttt{y>a+b}]\!] \cup exps(\texttt{a+1})) \downarrow \texttt{a}$
$[\![\texttt{x=a+b}]\!] = ([\![\texttt{a=a+1}]\!] \cup exps(\texttt{a+b})) \downarrow \texttt{x}$
$[\![exit]\!] = [\![\texttt{y>a+b}]\!]$

Using one of our fixed-point algorithms, we obtain the minimal solution:

$[\![entry]\!] = \emptyset$
$[\![\texttt{var x,y,z,a,b}]\!] = \emptyset$
$[\![\texttt{z=a+b}]\!] = \{\texttt{a+b}\}$
$[\![\texttt{y=a*b}]\!] = \{\texttt{a+b}, \texttt{a*b}\}$
$[\![\texttt{y>a+b}]\!] = \{\texttt{a+b}, \texttt{y>a+b}\}$
$[\![\texttt{a=a+1}]\!] = \emptyset$
$[\![\texttt{x=a+b}]\!] = \{\texttt{a+b}\}$
$[\![exit]\!] = \{\texttt{a+b}, \texttt{y>a+b}\}$

The expressions available at the program point *before* a node v can be computed from this solution as $JOIN(v)$. In particular, the solution confirms our previous observations about `a+b`. With this knowledge, an optimizing compiler could systematically transform the program into a (slightly) more efficient version:

```
var x,y,z,a,b,aplusb;
aplusb = a+b;
z = aplusb;
y = a*b;
while (y > aplusb) {
  a = a+1;
  aplusb = a+b;
  x = aplusb;
}
```

Exercise 5.31: Estimate the worst-case time complexity of available expressions analysis, assuming that the naive fixed-point algorithm is used.

5.6 Very Busy Expressions Analysis

An expression is *very busy* if it will definitely be evaluated again before its value changes. To approximate this property, we can use the same lattice and auxiliary functions as for available expressions analysis. For every CFG node v the variable $[\![v]\!]$ denotes the set of expressions that at the program point before the node definitely are busy.

An expression is very busy if it is evaluated in the current node or will be evaluated in all future executions unless an assignment changes its value. For this reason, the $JOIN$ is defined by

$$JOIN(v) = \bigcap_{w \in succ(v)} [\![w]\!]$$

and assignments are modeled using the following constraint rule:

X = E: $\quad [\![v]\!] = JOIN(v)\downarrow X \ \cup exps(E)$

No expressions are very busy at exit nodes:

$$[\![exit]\!] = \emptyset$$

The rules for the remaining nodes, include branch conditions and output statements, are the same as for available expressions analysis.

On the example program:

```
var x,a,b;
x = input;
a = x-1;
b = x-2;
while (x>0) {
  output a*b-x;
  x = x-1;
}
output a*b;
```

the analysis reveals that `a*b` is very busy inside the loop. The compiler can perform *code hoisting* and move the computation to the earliest program point where it is very busy. This would transform the program into this more efficient version:

```
var x,a,b,atimesb;
x = input;
a = x-1;
b = x-2;
atimesb = a*b;
while (x>0) {
  output atimesb-x;
  x = x-1;
}
output atimesb;
```

5.7 Reaching Definitions Analysis

The *reaching definitions* for a given program point are those assignments that may have defined the current values of variables. For this analysis we need a powerset lattice of all assignments (represented as CFG nodes) occurring in the program. For the example program from before:

```
var x,y,z;
x = input;
while (x>1) {
  y = x/2;
  if (y>3) x = x-y;
  z = x-4;
```

```
  if (z>0) x = x/2;
  z = z-1;
}
output x;
```

the lattice modeling abstract states becomes:

$$States = (2^{\{\texttt{x=input,y=x/2,x=x-y,z=x-4,x=x/2,z=z-1}\}}, \subseteq)$$

For every CFG node v the variable $[\![v]\!]$ denotes the set of assignments that may define values of variables at the program point after the node. We define

$$JOIN(v) = \bigcup_{w \in pred(v)} [\![w]\!]$$

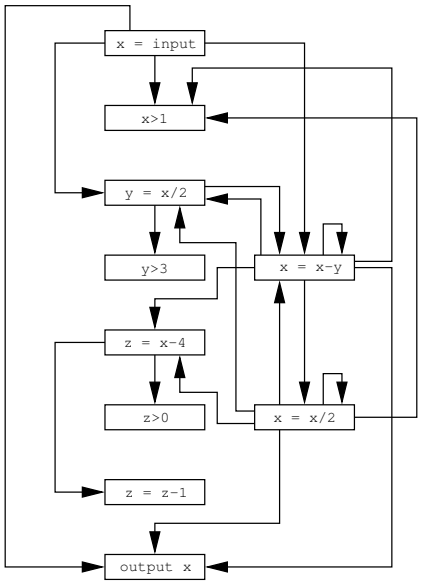
For assignments the constraint is:

$X = E$: $\quad [\![v]\!] = JOIN(v)\downarrow X \ \cup \{X = E\}$

where this time the $\downarrow X$ function removes all assignments to the variable X. For all other nodes we define:

$$[\![v]\!] = JOIN(v)$$

This analysis can be used to construct a *def-use graph*, which is like a CFG except that edges go from definitions (i.e. assignments) to possible uses. Here is the def-use graph for the example program:



The def-use graph is a further abstraction of the program and is the basis of widely used optimizations such as *dead code elimination* and *code motion*.

Exercise 5.32: Show that the def-use graph is always a subgraph of the transitive closure of the CFG.

5.8 Forward, Backward, May, and Must

As illustrated in the previous sections, a dataflow analysis is specified by providing the lattice and the constraint rules. Some patterns are emerging from the examples, which makes it possible to classify dataflow analyses in various ways.

A *forward* analysis is one that for each program point computes information about the *past* behavior. Examples of this are sign analysis and available expressions analysis. They can be characterized by the right-hand sides of constraints only depending on *predecessors* of the CFG node. Thus, the analysis essentially starts at the *entry* node and propagates information forward in the CFG. For such analyses, the $JOIN$ function is defined using $pred$, and dep (if using the work-list algorithm) can be defined by $succ$.

A *backward* analysis is one that for each program point computes information about the *future* behavior. Examples of this are live variables analysis and very busy expressions analysis. They can be characterized by the right-hand sides of constraints only depending on *successors* of the CFG node. Thus, the analysis starts at the *exit* node and moves backward in the CFG. For such analyses, the $JOIN$ function is defined using $succ$, and dep can be defined by $pred$.

The distinction between forward and backward applies to any flow-sensitive analysis. For analyses that are based on a powerset lattice, we can also distinguish between *may* and *must* analysis.

A *may* analysis is one that describes information that may possibly be true and, thus, computes an *over*-approximation. Examples of this are live variables analysis and reaching definitions analysis. They can be characterized by the lattice order being $\subseteq$ and constraint functions that use the $\cup$ operator to combine information.

Conversely, a *must* analysis is one that describes information that must definitely be true and, thus, computes an *under*-approximation. Examples of this are available expressions analysis and very busy expressions analysis. They can be characterized by the use of $\supseteq$ as lattice order and constraint functions that use $\cap$ to combine information.

Thus, our four examples that are based on powerset lattices show every possible combination, as illustrated by this diagram:

	Forward	*Backward*
May	Reaching Definitions	Live Variables
Must	Available Expressions	Very Busy Expressions

These classifications are mostly botanical, but awareness of them may provide inspiration for constructing new analyses.

Exercise 5.33: Which among the following analyses are distributive, if any?

(a) Available expressions analysis.

(b) Very busy expressions analysis.

(c) Reaching definitions analysis.

(d) Sign analysis.

(e) Constant propagation analysis.

Exercise 5.34: Let us design a *flow-sensitive type analysis* for TIP. In the simple version of TIP we focus on in this chapter, we only have integer values at runtime, but for the analysis we can treat the results of the comparison operators > and == as a separate type: boolean. The results of the arithmetic operators +, -, *, / can similarly be treated as type integer. As lattice for abstract states we choose

$$States = Vars \to 2^{\{\mathsf{integer},\mathsf{boolean}\}}$$

such that the analysis can keep track of the possible types for every variable.

(a) Specify constraint rules for the analysis.

(b) After analyzing a given program, how can we check using the computed abstract states whether the branch conditions in `if` and `while` statements are guaranteed to be booleans? Similarly, how can we check that the arguments to the arithmetic operators +, -, *, / are guaranteed to be integers? As an example, for the following program two warnings should be emitted:

```
main(a,b) {
  var x,y;
  x = a+b;
  if (x) { // warning: using integer as branch condition
    output 17;
  }
  y = a>b;
  return y+3; // warning: using boolean in addition
}
```

Exercise 5.35: Assume we want to build an optimizing compiler for TIP (without pointers and function calls). As part of this, we want to infer safely approximate the possible values for each variable to be able to pick appropriate runtime representations: bool (can represent only the two integer values 0 and 1), byte (8 bit signed integers), char (16 bit unsigned integers), int (32 bit signed integers), or bigint (any integer). Naturally, we do not want to waste space, so we prefer, for example, bit to int if we can guarantee that the value of the variable can only be 0 or 1.

As an extra feature, we introduce a cast operation in TIP: an expression of the form $(T)E$ where T is one of the five types and E is an expression. At runtime, a cast expression evaluates to the same value as E, except that it aborts program execution if the value does not fit into T.

(a) Define a suitable lattice for describing abstract states.

(b) Specify the constraint rules for your analysis.

(c) Write a small but nontrivial TIP program that gives rise to several different types, and argue briefly what result your analysis will produce for that program.

5.9 Initialized Variables Analysis

Let us try to define an analysis that ensures that variables are initialized (i.e. written to) before they are read. (A similar analysis is performed by Java compilers to check that every local variable has a definitely assigned value when any access of its value occurs.) This can be achieved by computing for every program point the set of variables that are guaranteed to be initialized. We need definite information, which implies a must analysis. Consequently, we choose as abstract state lattice the powerset of variables occurring in the given program, ordered by the superset relation. Initialization is a property of the past, so we need a forward analysis. This means that our constraints are phrased in terms of predecessors and intersections. On this basis, the constraint rules more or less give themselves.

Exercise 5.36: What is the $JOIN$ function for initialized variables analysis?

Exercise 5.37: Specify the constraint rule for assignments.

No other statements than assignments affect which variables are initialized, so the constraint rule for all other kinds of nodes is the same as, for example, in sign analysis (see page 41).

Using the results from initialized variables analysis, a programming error

detection tool could now check for every use of a variable that it is contained in the computed set of initialized variables, and emit a warning otherwise. A warning would be emitted for this trivial example program:

```
main() {
  var x;
  return x;
}
```

Exercise 5.38: Write a TIP program where such an error detection tool would emit a *spurious warning*. That is, in your program there are no reads from uninitialized variables in any execution but the initialized variables analysis is too imprecise to show it.

Exercise 5.39: An alternative way to formulate initialized variables analysis would be to use the following map lattice instead of the powerset lattice:

$$States = Vars \to Init$$

where $Init$ is a lattice with two elements $\{\mathsf{Initialized}, \mathsf{NotIninitialized}\}$.

(a) How should we order the two elements? That is, which one is $\top$ and which one is $\bot$?

(b) How should the constraint rule for assignments be modified to fit with this alternative lattice?

5.10 Transfer Functions

Observe that in all the analyses presented in this chapter, all constraint functions are of the form

$$[\![v]\!] = t_v(JOIN(v))$$

for some function $t_v : L \to L$ where L is the lattice modeling abstract states and $JOIN(v) = \bigsqcup_{w \in dep^{-1}(v)} [\![w]\!]$. The function t_v is called the *transfer function* for the CFG node v and specifies how the analysis models the statement at v as an abstract state transformer. For a forward analysis, which is the most common kind of dataflow analysis, the input to the transfer function represents the abstract state at the program point immediately before the statement, and its output represents the abstract state at the program point immediately after the statement (and conversely for a backward analysis). When specifying constraints for a dataflow analyses, it thus suffices to provide the transfer functions for all CFG nodes. As an example, in sign analysis where $L = Vars \to Sign$, the transfer function for assignment nodes $X = E$ is:

$$t_{X=E}(s) = s[X \mapsto eval(s, E)]$$

In the simple work-list algorithm, $JOIN(v) = \bigsqcup_{w \in dep^{-1}(v)} [\![w]\!]$ is computed in each iteration of the while-loop. However, often $[\![w]\!]$ has not changed since last time v was processed, so much of that computation may be redundant. (When we introduce inter-procedural analysis in Chapter 7, we shall see that $dep^{-1}(v)$ may become large.) We now present another work-list algorithm based on transfer functions that avoids some of that redundancy. With this algorithm, for a forward analysis each variable x_i denotes the abstract state for the program point *before* the corresponding CFG node v_i, in contrast to the other fixed-point solvers we have seen previously where x_i denotes the abstract state for the program point *after* v_i (and conversely for a backward analysis).

procedure TRANSFERWORKLISTALGORITHM($t_1, \ldots, t_n$)
  $(x_1, \ldots, x_n) := (\bot, \ldots, \bot)$
  $W := \{v_1, \ldots, v_n\}$
  while $W \neq \emptyset$ **do**
    $v_i := W$.**removeNext**()
    $y := t_{v_i}(x_i)$
    for each $v_j \in dep(v_i)$ **do**
      $z := x_j \sqcup y$
      if $x_j \neq z$ **then**
        $x_j := z$
        W.**add**(v_j)
      end if
    end for
  end while
  return $(x_1, \ldots, x_n)$
end procedure

In each iteration of the while-loop, the transfer function of the current node v_i is applied, and the resulting abstract state is propagated to all dependencies. Those that change are added to the work-list.

Exercise 5.40: Prove that TRANSFERWORKLISTALGORITHM computes the same solution as the other fixed-point solvers.

5.11 Interval Analysis

An *interval analysis* computes for every integer variable a lower and an upper bound for its possible values. Intervals are interesting analysis results, since sound answers can be used for optimizations and bug detection related to array bounds checking, numerical overflows, and integer representations.

This example involves a lattice of infinite height, and we must use a special technique described in Section 5.12 to to ensure convergence toward a fixed-point.

The lattice describing a single abstract value is defined as

$$\mathit{Interval} = \mathit{lift}(\{[l,h] \mid l,h \in N \ \wedge \ l \leq h\})$$

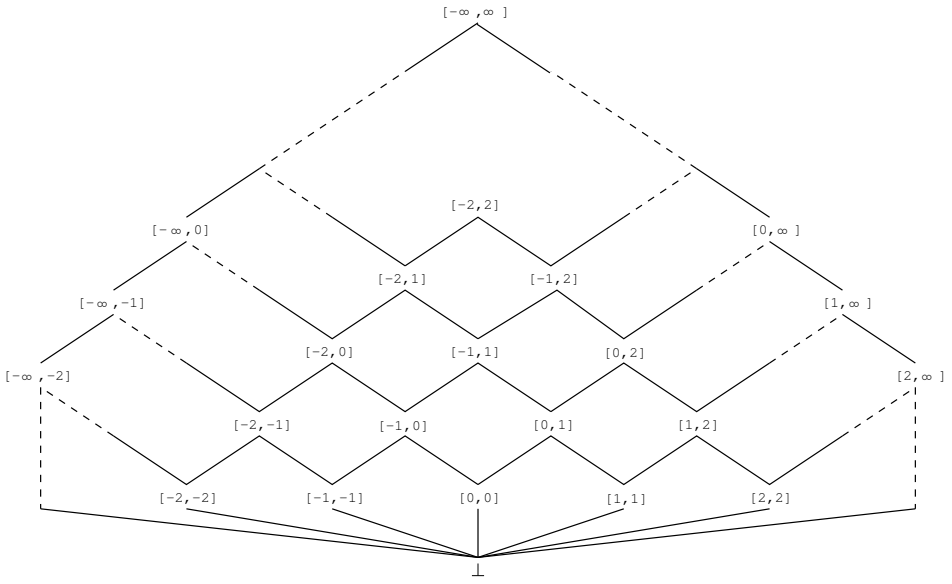
where

$$N = \{-\infty, \ldots, -2, -1, 0, 1, 2, \ldots, \infty\}$$

is the set of integers extended with infinite endpoints and the order on intervals is defined by inclusion:

$$[l_1,h_1] \sqsubseteq [l_2,h_2] \ \Leftrightarrow \ l_2 \leq l_1 \wedge h_1 \leq h_2$$

This lattice looks as follows:



It is clear that we do not have a lattice of finite height, since it contains for example this infinite chain:

$$[0,0] \sqsubseteq [0,1] \sqsubseteq [0,2] \sqsubseteq [0,3] \sqsubseteq [0,4] \sqsubseteq [0,5] \ldots$$

This carries over to the lattice for abstract states:

$$\mathit{States} = \mathit{Vars} \rightarrow \mathit{Interval}$$

Before we specify the constraint rules, we define a function eval that performs an abstract evaluation of expressions:

$\mathit{eval}(\sigma, X) = \sigma(X)$
$\mathit{eval}(\sigma, I) = [I, I]$
$\mathit{eval}(\sigma, E_1 \ \mathsf{op} \ E_2) = \widehat{\mathsf{op}}(\mathit{eval}(\sigma, E_1), \mathit{eval}(\sigma, E_2))$

The abstract arithmetical operators all are defined by:

$$\widehat{\mathsf{op}}([l_1,h_1],[l_2,h_2]) = [\min_{x\in[l_1,h_1],y\in[l_2,h_2]} x \,\mathsf{op}\, y, \max_{x\in[l_1,h_1],y\in[l_2,h_2]} x \,\mathsf{op}\, y]$$

For example, $\widehat{+}([1,10],[-5,7]) = [1-5, 10+7] = [-4,17]$.

Exercise 5.41: Explain why the definition of $eval$ given above is a conservative approximation compared to evaluating TIP expressions concretely. Give an example of how the definition could be modified to make the analysis more precise (and still sound).

Exercise 5.42: This general definition of $\widehat{\mathsf{op}}$ looks simple in math, but it is nontrivial to implement it efficiently. Your task is to write pseudo-code for an implementation of the abstract greater-than operator $\widehat{>}$. (To be usable in practice, the execution time of your implementation should be less than linear in the input numbers!)

The $JOIN$ function is the usual one for forward analyses:

$$JOIN(v) = \bigsqcup_{w\in pred(v)} [\![w]\!]$$

We can now specify the constraint rule for assignments:

X = E: $\quad [\![v]\!] = JOIN(v)[X \mapsto eval(JOIN(v), E)]$

For all other nodes the constraint is the trivial one:

$$[\![v]\!] = JOIN(v)$$

Exercise 5.43: Argue that the constraint functions are monotone.

The interval analysis lattice has infinite height, so applying the naive fixed-point algorithms may never terminate: for the lattice L^n, the sequence of approximants

$$f^i(\bot,\ldots,\bot)$$

need never converge. A powerful technique to address this kind of problem is introduced in the next section.

Exercise 5.44: Give an example of a TIP program where none of the fixed-point algorithms terminate for the interval analysis as presented above.

5.12 Widening and Narrowing

To obtain convergence of the interval analysis presented in Section 5.11 we shall use a technique called *widening*. This technique generally works for any analysis

that can expressed using monotone equation systems, but it is typically used in flow-sensitive analyses with infinite-height lattices.

Let $f : L \to L$ denote the function from the fixed-point theorem and the naive fixed-point algorithm (Section 4.4). A particularly simple form of widening, which often suffices in practice, introduces a function $w : L \to L$ so that the sequence

$$(w \circ f)^i(\bot) \quad \text{for } i = 0, 1, \ldots$$

is guaranteed to converge on a fixed-point that is at larger than or equal to each approximant $f^i(\bot)$ of the naive fixed-point algorithm and thus represents sound information about the program. To ensure this property, it suffices that w is monotone and extensive (see Exercise 4.14), and that the image $w(L) = \{y \in L \mid \exists x \in L : y = w(x)\}$ has finite height. The fixed-point algorithms can easily be adapted to use widening by applying w in each iteration.

The widening function w will intuitively coarsen the information sufficiently to ensure termination. For our interval analysis, w is defined pointwise down to single intervals. It operates relatively to a fixed finite subset $B \subset N$ that must contain $-\infty$ and ∞. Typically, B could be seeded with all the integer constants occurring in the given program, but other heuristics could also be used. On a single interval we have

$$w([l, h]) = [max\{i \in B \mid i \leq l\}, min\{i \in B \mid h \leq i\}]$$

which finds the best fitting interval among the ones that are allowed.

Exercise 5.45: Show that interval analysis with widening, using this definition of w, always terminates and yields a solution that is a safe approximation of the ideal result.

Widening generally shoots above the target, but a subsequent technique called *narrowing* may improve the result. If we define

$$\mathit{fix} = \bigsqcup f^i(\bot) \qquad \mathit{fixw} = \bigsqcup (w \circ f)^i(\bot)$$

then we have $\mathit{fix} \sqsubseteq \mathit{fixw}$. However, we also have that $\mathit{fix} \sqsubseteq f(\mathit{fixw}) \sqsubseteq \mathit{fixw}$, which means that a subsequent application of f may improve our result and still produce sound information. This technique, called *narrowing*, may in fact be iterated arbitrarily many times.

Exercise 5.46: Show that $\forall i : \mathit{fix} \sqsubseteq f^{i+1}(\mathit{fixw}) \sqsubseteq f^i(\mathit{fixw}) \sqsubseteq \mathit{fixw}$.

An example will demonstrate the benefits of these techniques. Consider this program:

```
y = 0; x = 7; x = x+1;
while (input) {
  x = 7;
  x = x+1;
```

```
    y = y+1;
}
```

Without widening, the analysis will produce the following diverging sequence of approximants for the program point after the loop:

$[\mathtt{x} \mapsto \bot, \mathtt{y} \mapsto \bot]$
$[\mathtt{x} \mapsto [8,8], \mathtt{y} \mapsto [0,1]]$
$[\mathtt{x} \mapsto [8,8], \mathtt{y} \mapsto [0,2]]$
$[\mathtt{x} \mapsto [8,8], \mathtt{y} \mapsto [0,3]]$
$\vdots$

If we apply widening, based on the set $B = \{-\infty, 0, 1, 7, \infty\}$ seeded with the constants occurring in the program, then we obtain a converging sequence:

$[\mathtt{x} \mapsto \bot, \mathtt{y} \mapsto \bot]$
$[\mathtt{x} \mapsto [7,\infty], \mathtt{y} \mapsto [0,1]]$
$[\mathtt{x} \mapsto [7,\infty], \mathtt{y} \mapsto [0,7]]$
$[\mathtt{x} \mapsto [7,\infty], \mathtt{y} \mapsto [0,\infty]]$

However, the result for `x` is discouraging. Fortunately, a few iterations of narrowing quickly improve the result:

$[\mathtt{x} \mapsto [8,8], \mathtt{y} \mapsto [0,\infty]]$

This result is really the best we could hope for, for this program. For that reason, further narrowing has no effect. Note that the decreasing sequence

$$\mathit{fix}w \sqsupseteq f(\mathit{fix}w) \sqsupseteq f^2(\mathit{fix}w) \sqsupseteq f^3(\mathit{fix}w) \ldots$$

is not guaranteed to converge, so heuristics must determine how many times to apply narrowing.

Chapter 6

Path Sensitivity

Until now, we have ignored the values of conditions by simply treating `if`- and `while`-statements as a nondeterministic choice between the two branches. This is called a *path insensitive* analysis as it does not distinguish different paths that lead to a given program point. This technique fails to include some information that could potentially be used in a static analysis. Consider for example the following program:

```
x = input;
y = 0;
z = 0;
while (x > 0) {
  z = z+x;
  if (17 > y) { y = y+1; }
  x = x-1;
}
```

The previous interval analysis (with widening) will conclude that after the `while`-loop, the variable `x` is in the interval $[-\infty, \infty]$, `y` is in the interval $[0, \infty]$, and `z` is in the interval $[-\infty, \infty]$. However, in view of the conditionals being used, this result is too pessimistic.

6.1 Assertions

To exploit the available information, we shall extend the language with an artificial statement, `assert(`E`)`, where E is a boolean expression. This statement will abort execution at runtime if E is false and otherwise have no effect, however, we shall only insert it at places where E is guaranteed to be true. In the interval analysis, the constraints for these new statement will narrow the intervals for the various variables by exploiting information in conditionals.

For the example program, the meanings of the conditionals can be encoded by the following program transformation:

```
x = input;
y = 0;
z = 0;
while (x > 0) {
  assert(x > 0);
  z = z+x;
  if (17 > y) { assert(17 > y); y = y+1; }
  x = x-1;
}
assert(!(x > 0));
```

(We here also extend TIP with a unary negation operator `!`.) It is always safe to ignore the `assert` statements, which amounts to this trivial constraint rule:

`assert(`E`)`: $\quad [\![v]\!] = JOIN(v)$

With that constraint rule, no extra precision is gained. It requires insight into the specific static analysis to define nontrivial and sound constraints for assertions.

For the interval analysis, extracting the information carried by general conditions, or *predicates*, such as $E_1 > E_2$ or $E_1 == E_2$ relative to the lattice elements is complicated and in itself an area of considerable study. For simplicity, let us consider conditions only of the two kinds $X > E$ and $E > X$. The former kind of assertion can be handled by the constraint rule

`assert(`$X > E$`)`: $\quad [\![v]\!] = JOIN(v)[X \mapsto gt(JOIN(v)(X), eval(JOIN(v), E))]$

where gt models the greater-than operator:

$$gt([l_1, h_1], [l_2, h_2]) = [l_1, h_1] \sqcap [l_2, \infty]$$

Exercise 6.1: Argue that this constraint for `assert` is sound and monotone.

Exercise 6.2: Specify a constraint rule for `assert(`$E > X$`)`.

Negated conditions are handled in similar fashions, and all other conditions are given the trivial constraint by default.

With this refinement, the interval analysis of the above example will conclude that after the `while`-loop the variable `x` is in the interval $[-\infty, 0]$, `y` is in the interval $[0, 17]$, and `z` is in the interval $[0, \infty]$.

Exercise 6.3: Discuss how more conditions may be given nontrivial constraints for `assert` to improve analysis precision further.

6.2 Branch Correlations

The use of `assert` statements at conditional branches provides a simple kind of path sensitivity called *control sensitivity* , however it is insufficient for reasoning about correlations of branches in programs. Here is a typical example:

```
if (condition) {
  open();
  flag = 1;
} else {
  flag = 0;
}
...
if (flag) {
  close();
}
```

We here assume that `open` and `close` are built-in functions for opening and closing a specific file. The file is initially closed, `condition` is some complex expression, and the "..." consists of statements that do not call `open` or `close` or modify `flag`. We wish to design an analysis that can check that `close` is only called if the file is currently open.

As a starting point, we use this lattice for modeling the open/closed status of the file:

$$L = (2^{\{\mathsf{open},\mathsf{closed}\}}, \subseteq)$$

For every CFG node v the variable $[\![v]\!]$ denotes the possible status of the file at the program point after the node. For `open` and `close` statements the constraints are:

$$[\![\texttt{open()}]\!] = \{\mathsf{open}\}$$

$$[\![\texttt{close()}]\!] = \{\mathsf{closed}\}$$

For the entry node, we define:

$$[\![entry]\!] = \{\mathsf{closed}\}$$

and for every other node, which does not modify the file status, the constraint is simply

$$[\![v]\!] = JOIN(v)$$

where $JOIN$ is defined as usual for a forward, may analysis:

$$JOIN(v) = \bigcup_{w \in pred(v)} [\![w]\!]$$

In the example program, the `close` function is clearly called if and only if `open` is called, but the current analysis fails to discover this.

Exercise 6.4: Write the constraints being produced for the example program and show that the solution for $[\![\texttt{flag}]\!]$ (the node for the last `if` condition) is $\{\mathsf{open}, \mathsf{closed}\}$.

Arguing that the program has the desired property obviously involves the `flag` variable, which the lattice above ignores. So, we can try with a slightly more sophisticated lattice – a product lattice that keeps track of both the status of the file and the value of the flag:

$$L' = (2^{\{\mathsf{open},\mathsf{closed}\}}, \subseteq) \times (2^{\{\texttt{flag}=0,\texttt{flag}\neq 0\}}, \subseteq)$$

Additionally, we insert `assert` statements to model the conditionals:

```
if (condition) {
  assert(condition);
  open();
  flag = 1;
} else {
  assert(!condition);
  flag = 0;
}
...
if (flag) {
  assert(flag);
  close();
} else {
  assert(!flag);
}
```

This is still insufficient, though. At the program point after the first `if-else` statement, the analysis only knows that `open` *may* have been called and `flag` *may* be 0.

Exercise 6.5: Specify the constraints that fit with the L' lattice. Then show that the analysis produces the lattice element $(2^{\{\mathsf{open},\mathsf{closed}\}}, 2^{\{\texttt{flag}=0,\texttt{flag}\neq 0\}})$ for the first node after the first `if-else` statement.

The present analysis is also called an *independent attribute analysis* as the abstract value of the file is independent of the abstract value of the boolean flag. What we need is a *relational* analysis that can keep track of relations between variables. This can be achieved by generalizing the analysis to maintain *multiple* abstract states per program point. If L is the original lattice as defined above, we replace it by the map lattice

$$L'' = Paths \to L$$

where $Paths$ is a finite set of *path contexts*. A path context is here a predicate over the program state. (For instance, a condition expression in TIP defines such

a predicate.) In general, each statement is then analyzed in $|Paths|$ different path contexts, each describing a set of paths that lead to the statement. For the example above, we can use $Paths = \{\texttt{flag} = 0, \texttt{flag} \neq 0\}$.

The constraints for `open`, `close`, and *entry* can now be defined as follows.[1]

`open()`: $[\![v]\!] = \lambda p.\{\mathsf{open}\}$

`close()`: $[\![v]\!] = \lambda p.\{\mathsf{closed}\}$

entry: $[\![v]\!] = \lambda p.\{\mathsf{closed}\}$

The constraints for assignments make sure that `flag` gets special treatment:

`flag = 0`: $[\![v]\!] = [\texttt{flag} = 0 \mapsto \bigcup_{p \in Paths} JOIN(v)(p),\ \texttt{flag} \neq 0 \mapsto \emptyset]$

`flag = I`: $[\![v]\!] = [\texttt{flag} \neq 0 \mapsto \bigcup_{p \in Paths} JOIN(v)(p),\ \texttt{flag} = 0 \mapsto \emptyset]$

`flag = E`: $[\![v]\!] = \lambda q. \bigcup_{p \in Paths} JOIN(v)(p)$

Here, I is an integer constant other than `0` and E is a non-integer-constant expression. The definition of $JOIN$ follows from the lattice structure and from the analysis being forward:

$$JOIN(v)(p) = \bigcup_{w \in pred(v)} [\![w]\!](p)$$

The constraint for the case `flag = 0` models the fact that `flag` is definitely 0 after the statement, so the open/closed information is obtained from the predecessors, independent of whether `flag` was 0 or not before the statement. Also, the open/closed information is set to the bottom element $\emptyset$ for $\texttt{flag} \neq 0$ because that path context is infeasible at the program point after `flag = 0`. The constraint for `flag = I` is dual, and the last constraint covers the cases where `flag` is assigned an unknown value.

For `assert` statements, we also give special treatment to `flag`:

`assert(flag)`: $[\![v]\!] = [\texttt{flag} \neq 0 \mapsto JOIN(v)(\texttt{flag} \neq 0),\ \texttt{flag} = 0 \mapsto \emptyset]$

Notice the small but important difference compared to the constraint for `flag = 1` statements. As before, the case for negated expressions is similar.

Exercise 6.6: Give an appropriate constraint for `assert(!flag)`.

Finally, for any other node v, including other `assert` statements, the constraint keeps the dataflow information for different path contexts apart but otherwise simply propagates the information from the predecessors in the CFG:

$$[\![v]\!] = \lambda p.JOIN(v)(p)$$

[1] We here use the lambda abstration notation to denote a function: if $f = \lambda x.e$ then $f(x) = e$. Thus, $\lambda p.\{\mathsf{open}\}$ is the function that returns $\{\mathsf{open}\}$ for any input p.

Although this is sound, we could make more precise constraints for `assert` nodes by recognizing other patterns that fit into the abstraction given by the lattice.

For our example program, the following constraints are generated:

$[\![\textit{entry}]\!] = \lambda p.\{\mathsf{closed}\}$
$[\![\texttt{condition}]\!] = [\![\textit{entry}]\!]$
$[\![\texttt{assert(condition)}]\!] = [\![\texttt{condition}]\!]$
$[\![\texttt{open()}]\!] = \lambda p.\{\mathsf{open}\}$
$[\![\texttt{flag = 1}]\!] = \left[\texttt{flag} \neq 0 \mapsto \bigcup_{p \in Paths} [\![\texttt{open()}]\!](p),\ \texttt{flag} = 0 \mapsto \emptyset\right]$
$[\![\texttt{assert(!condition)}]\!] = [\![\texttt{condition}]\!]$
$[\![\texttt{flag = 0}]\!] = \left[\texttt{flag} = 0 \mapsto \bigcup_{p \in Paths} [\![\texttt{assert(!condition)}]\!](p),\ \texttt{flag} \neq 0 \mapsto \emptyset\right]$
$[\![\texttt{...}]\!] = \lambda p.\big([\![\texttt{flag = 1}]\!](p) \cup [\![\texttt{flag = 0}]\!](p)\big)$
$[\![\texttt{flag}]\!] = [\![\texttt{...}]\!]$
$[\![\texttt{assert(flag)}]\!] = [\texttt{flag} \neq 0 \mapsto [\![\texttt{flag}]\!](\texttt{flag} \neq 0),\ \texttt{flag} = 0 \mapsto \emptyset]$
$[\![\texttt{close()}]\!] = \lambda p.\{\mathsf{closed}\}$
$[\![\texttt{assert(!flag)}]\!] = [\texttt{flag} = 0 \mapsto [\![\texttt{flag}]\!](\texttt{flag} = 0),\ \texttt{flag} \neq 0 \mapsto \emptyset]$
$[\![\textit{exit}]\!] = \lambda p.\big([\![\texttt{close()}]\!](p) \cup [\![\texttt{assert(!flag)}]\!](p)\big)$

The minimal solution is, for each $[\![v]\!](p)$:

	$\texttt{flag} = 0$	$\texttt{flag} \neq 0$
$[\![\textit{entry}]\!]$	{closed}	{closed}
$[\![\texttt{condition}]\!]$	{closed}	{closed}
$[\![\texttt{assert(condition)}]\!]$	{closed}	{closed}
$[\![\texttt{open()}]\!]$	{open}	{open}
$[\![\texttt{flag = 1}]\!]$	$\emptyset$	{open}
$[\![\texttt{assert(!condition)}]\!]$	{closed}	{closed}
$[\![\texttt{flag = 0}]\!]$	{closed}	$\emptyset$
$[\![\texttt{...}]\!]$	{closed}	{open}
$[\![\texttt{flag}]\!]$	{closed}	{open}
$[\![\texttt{assert(flag)}]\!]$	$\emptyset$	{open}
$[\![\texttt{close()}]\!]$	{closed}	{closed}
$[\![\texttt{assert(!flag)}]\!]$	{closed}	$\emptyset$
$[\![\textit{exit}]\!]$	{closed}	{closed}

The analysis produces the lattice element $[\texttt{flag} = 0 \mapsto \{\mathsf{closed}\}, \texttt{flag} \neq 0 \mapsto \{\mathsf{open}\}]$ for the program point after the first `if-else` statement. The constraint for the `assert(flag)` statement will eliminate the possibility that the file is closed at that point. This ensures that `close` is only called if the file is open, as desired.

Exercise 6.7: For the present example, the basic lattice L is a defined as a powerset of a finite set A. Show that $Paths \to 2^A$ is isomorphic to $2^{Paths \times A}$. (This explains why such analyses are called *relational*: each element of $2^{Paths \times A}$ is a (binary) relation between $Paths$ and A.)

Exercise 6.8: Describe a variant of the example program above where the present analysis would be improved if combining it with constant propagation.

In general, the program analysis designer is left with the choice of $Paths$. Often, $Paths$ consists of combinations of predicates that appear in conditionals in the program. This quickly results in an exponential blow-up: for k predicates, each statement may need to be analyzed in 2^k different path contexts. In practice, however, there is usually much redundancy in these analysis steps. Thus, in addition to the challenge of reasoning about the `assert` predicates relative to the lattice elements, it requires a considerable effort to avoid too many redundant computations in path sensitive analysis. One approach is *iterative refinement* where $Paths$ is initially a single universal path context, which is then iteratively refined by adding relevant predicates until either the desired properties can be established or disproved or the analysis is unable to select relevant predicates and hence gives up.

Exercise 6.9: Assume that we change the rule for `open` from

$$[\![\texttt{open()}]\!] = \lambda p.\{\texttt{open}\}$$

to

$$[\![\texttt{open()}]\!] = \lambda p.\text{ if } JOIN(v)(p) = \emptyset \text{ then } \emptyset \text{ else } \{\texttt{open}\}$$

Argue that this is sound and for some programs more precise than the original rule.

Exercise 6.10: The following is a variant of the previous example program:

```
if (condition) {
  flag = 1;
} else {
  flag = 0;
}
...
if (flag) {
  open();
}
...
if (flag) {
  close();
}
```

(Again, assume that "`...`" are statements that do not call `open` or `close` or modify `flag`.) Is the path sensitive analysis described in this section capable of showing also for this program that `close` is called only if the file is open?

Exercise 6.11: Construct yet another variant of the `open/close` example program where the desired property can only be established with a choice of $Paths$ that includes a predicate that does *not* occur as a conditional expression in the program source. (Such a program may be challenging to handle with iterative refinement techniques.)

Chapter 7

Interprocedural Analysis

So far, we have only analyzed the body of a single function, which is called an *intraprocedural* analysis. We now consider *interprocedural* analysis of whole programs containing multiple functions and function calls.

7.1 Interprocedural Control Flow Graphs

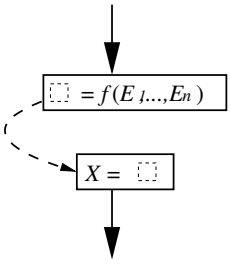
We use the subset of the TIP language containing functions, but still ignore pointers. As we shall see, the CFG for an entire program is then quite simple to obtain. It becomes more complicated when adding function pointers, which we discuss in Chapter 8.

First we construct the CFGs for all individual function bodies as usual. All that remains is then to glue them together to reflect function calls properly. We need to take care of parameter passing, return values, and values of local variables across calls. For simplicity we assume that all function calls are performed in connection with assignments:

X = f`(`E_1`,` ... `,` E_n`);`

Exercise 7.1: Show how any program can be normalized (cf. Section 2.3) to have this form.

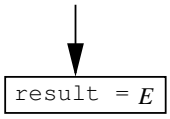
In the CFG, we represent each function call statement using *two* nodes: a *call node* representing the connection from the caller to the entry of `f`, and an *after-call node* where execution resumes after returning from the exit of `f`:



Next, we represent each `return` statement

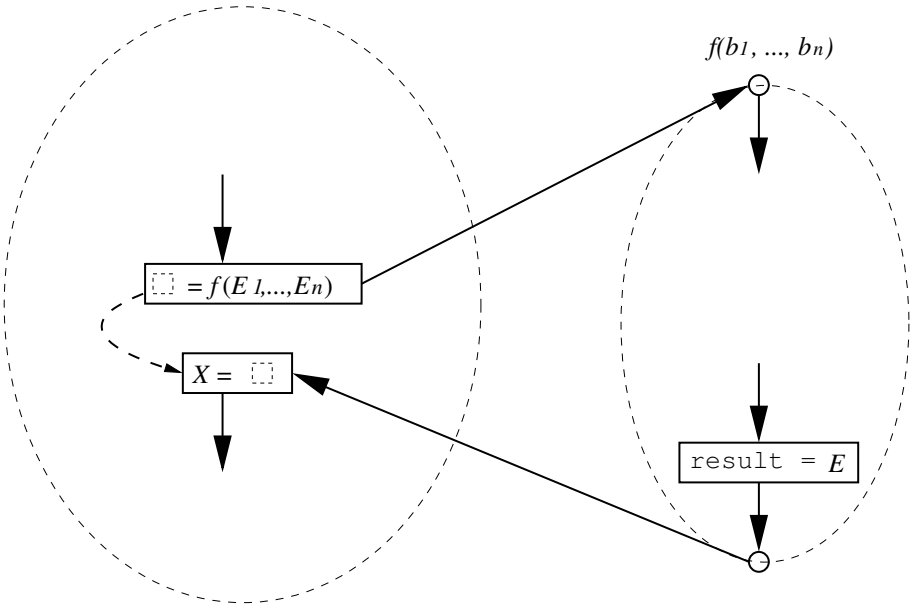
```
return E;
```

as an assignment using a special variable named `result`:



As discussed in Section 2.5, CFGs can be constructed such that there is always a unique entry node and a unique exit node for each function.

We can now glue together the caller and the callee as follows:

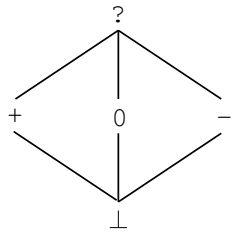


The connection between the call node and its after-call node is represented by a special edge (not in *succ* and *pred*), which we need for propagating abstract values for local variables of the caller.

With this interprocedural CFG in place, we can apply the monotone framework. Examples are given in the following sections.

Exercise 7.2: How many edges may the interprocedural CFG contain in a program with n CFG nodes?

Recall the intraprocedural sign analysis from Sections 4.1 and 5.1. That analysis models values with the lattice Sign:



and abstract states are represented by the map lattice $\mathit{States} = \mathit{Vars} \to \mathit{Sign}$. For any program point, the abstract state only provides information about variables that are in scope; all other variables can be set to $\bot$.

To make the sign analysis interprocedural, we define constraints for function entries and exits. For an entry node v of a function $f\texttt{(}b_1\texttt{, }\ldots\texttt{, }b_n\texttt{)}$ we consider the abstract states for all callers $\mathit{pred}(v)$ and model the passing of parameters:

$$[\![v]\!] = \bigsqcup_{w \in \mathit{pred}(v)} s_w$$

where[1]

$$s_w = \bot[b_1 \mapsto \mathit{eval}([\![w]\!], E_1^w), \ldots, b_n \mapsto \mathit{eval}([\![w]\!], E_n^w)]$$

where E_i^w is the i'th argument at the call node w. As discussed in Section 4.4, constraints can be expressed using inequations instead of equations. The constraint rule above can be reformulated as follows, where v is a function entry node v and $w \in \mathit{pred}(v)$ is a caller:

$$[\![v]\!] \sqsupseteq s_w$$

Intuitively, this shows how information flows from the call node (the right-hand-side of $\sqsupseteq$) to the function entry node (the left-hand-side of $\sqsupseteq$).

Exercise 7.3: Explain why these two formulations of the constraint rule for function entry nodes are equivalent.

For the entry node v of the `main` function with parameters $b_1, \ldots, b_n$ we have this special rule that models the fact that `main` is implicitly called with unknown arguments:

$$[\![v]\!] = \bot[b_1 \mapsto \texttt{?}, \ldots, b_n \mapsto \texttt{?}]$$

[1] In this expression, $\bot$ denotes the bottom element of the $\mathit{Vars} \to \mathit{Sign}$, that is, it maps every variable to the bottom element of Sign.

For an after-call node v that stores the return value in the variable X and where v' is the accompanying call node and $w \in pred(v)$ is the function exit node, the dataflow can be modeled by the following constraint:

$$[\![v]\!] = [\![v']\!][X \mapsto [\![w]\!](\texttt{result})]$$

The constraint obtains the abstract values of the local variables from the call node v' and the abstract value of `result` from w.

With this approach, no constraints are needed for call nodes and exit nodes. In a backward analysis, one would consider the call nodes and the function exit nodes rather than the function entry nodes and the after-call nodes. Also notice that we exploit the fact that the variant of the TIP language we use in this chapter does not have global variables, a heap, nested functions, or higher-order functions.

Exercise 7.4: Write and solve the constraints that are generated by the interprocedural sign analysis for the following program:

```
inc(a) {
  return a+1;
}

main() {
  var x,y;
  x = inc(17);
  y = inc(87);
  return x+y;
}
```

Exercise 7.5: Assume we extend TIP with *global variables*. Such variables are declared before all functions and their scope covers all functions. Write a TIP program with global variables that is analyzed incorrectly (that is, unsoundly) with the current analysis. Then show how the constraint rules above should be modified to accommodate this language feature.

Function entry nodes may have many predecessors, and similarly, function exit nodes may have many successors. For this reason, using algorithms like TransferWorkListAlgorithm (Section 5.10) are often preferred for interprocedural dataflow analysis.

Exercise 7.6: For the interprocedural sign analysis, how can we choose $dep(v)$ when v is a call node, an after-call node, a function entry node, or a function exit node?

7.2 Context Sensitivity

The approach to interprocedural analysis as presented in the previous sections is called *context insensitive*, because it does not distinguish between different calls to the same function. As an example, consider the sign analysis applied to this program:

```
f(z) {
  return z*42;
}

main() {
  var x,y;
  x = f(0);   // call 1
  y = f(87);  // call 2
  return x + y;
}
```

Due to the first call to `f` the parameter `z` may be 0, and due to the second call it may be a positive number, so in the abstract state at the entry of `f`, the abstract value of `z` is `?`. That value propagates through the body of `f` and back to the callers, so both `x` and `y` also become `?`. This is an example of dataflow along *interprocedurally invalid paths*: according to the analysis constraints, dataflow from one call node propagates through the function body and returns not only at the matching after-call node but at all after-call nodes. Although the analysis is still sound, the resulting loss of precision may be unacceptable.

A naive solution to this problem is to use function cloning. In this specific example we could clone `f` and let the two calls invoke different but identical functions. A similar effect would be obtained by inlining the function body at each call. More generally this may, however, increase the program size significantly, and in case of (mutually) recursive functions it would result in infinitely large programs. As we shall see next, we can instead encode the relevant information to distinguish the different calls by the use of more expressive lattices, much like the path-sensitivity approach in Chapter 6.

As discussed in the previous section, a basic context-insensitive dataflow analysis can be expressed using a lattice $States^n$ where $States$ is the lattice describing abstract states and $n = |Nodes|$ (or equivalently, using a lattice $Nodes \to States$). Context-sensitive analysis instead uses a lattice of the form

$$\big(Contexts \to \mathit{lift}(States)\big)^n$$

(or equivalently, $Contexts \to (\mathit{lift}(States))^n$ or $Nodes \to Contexts \to \mathit{lift}(States)$ or $Contexts \times Nodes \to \mathit{lift}(States)$) where $Contexts$ is a set of *call contexts*. The reason for using the lifted sub-lattice $\mathit{lift}(States)$ (as defined in Section 4.3) is that $Contexts$ may be large so we only want to infer abstract states for call contexts that may be feasible. The bottom element of $\mathit{lift}(States)$, denoted unreachable, is

used for call contexts that are unreachable from the program entry. (Of course, if $States$ already provides similar information, we do not need the lifted version.)

In the following sections we present different ways of choosing the set of call contexts. A trivial choice is to let $Contexts$ be a singleton set, which amounts to context-insensitive analysis. Another extreme we shall investigate is to to pick $Contexts = States$, which allows *full* context sensitivity.

Dataflow for CFG nodes that do not involve function calls and returns is modeled as usual, except that we now have an abstract state (or the extra lattice element unreachable) for each call context. This means that the constraint variables now range over $Contexts \to lift(States)$ rather than just $States$. For example, the constraint rule for assignments X=E in intraprocedural sign analysis from Section 5.1,

$$X = E\text{:} \qquad [\![v]\!] = JOIN(v)[X \mapsto eval(JOIN(v), E)]$$

becomes

$$X = E\text{:} \qquad [\![v]\!](c) = \begin{cases} s[X \mapsto eval(s, E)] & \text{if } s = JOIN(v, c) \in States \\ \mathsf{unreachable} & \text{if } JOIN(v, c) = \mathsf{unreachable} \end{cases}$$

where

$$JOIN(v, c) = \bigsqcup_{w \in pred(v)} [\![w]\!](c)$$

to match the new lattice with context sensitivity. Note that information for different call contexts is kept apart, and that the reachability information is propagated along. How to model the dataflow at call nodes, after-call nodes, function entry nodes, and function exit nodes depends on the context sensitivity strategy, as described in the following sections.

7.3 Context Sensitivity with Call Strings

Let $Calls$ be the set of call nodes in the CFG. The *call string* approach to context sensitivity defines[2]

$$Contexts = Calls^{\leq k}$$

where k is a positive integer. With this choice of call contexts, we can obtain a similar effect as function cloning or inlining, but without actually changing the CFG. The idea is that a tuple $(c_1, c_2, \ldots, c_m) \in Calls^{\leq k}$ identifies the topmost m call sites on the call stack. If $(e_1, \ldots, e_n) \in (Contexts \to States)^n$ is a lattice element, then $e_i(c_1, c_2, \ldots, c_m)$ provides an abstract state that approximates the runtime states that may appear at the i'th CFG node, assuming that the function containing that node was called from c_1, and the function containing c_1 was called from c_2, etc.

[2]We here use the notation $A^{\leq k}$ meaning the set of tuples of k or fewer elements from the set A, or more formally: $A^{\leq k} = \bigcup_{i=0,\ldots,k} A^i$. The symbol ϵ denotes the empty tuple.

The worst-case complexity of the analysis is evidently affected by the choice of k.

Exercise 7.7: What is the height of the lattice $(Contexts \to States)^n$ when $Contexts = Calls^{\leq k}$ and $States = Vars \to Sign$, expressed in terms of k (the call string bound), $n = |Nodes|$, and $b = |Vars|$?

To demonstrate the call string approach we again consider sign analysis applied to the program from Section 7.2. Let c_1 and c_2 denote the two call nodes in the `main` function in the program. For simplicity, we focus on the case $k = 1$, meaning that $Contexts = \{\epsilon, c_1, c_2\}$, so the analysis only tracks the top-most call site. When execution is initiated at the `main` function, the current context is described by the empty call string ϵ. We can now define the analysis constraints such that, in particular, at the entry of the function `f`, we obtain the lattice element

$$\begin{array}{l}[\epsilon \mapsto \mathsf{unreachable},\\ \ c_1 \mapsto [\mathtt{x} \mapsto \bot, \mathtt{y} \mapsto \bot, \mathtt{z} \mapsto \mathtt{0}],\\ \ c_2 \mapsto [\mathtt{x} \mapsto \bot, \mathtt{y} \mapsto \bot, \mathtt{z} \mapsto \mathtt{+}]]\end{array}$$

which has different abstract values for `z` depending on the caller. Notice that the information for the context ϵ is unreachable, since `f` is not the main function but is always executed from c_1 or c_2.

The constraint rule for an entry node v of a function $f\,\mathtt{(}b_1\mathtt{,}\ \ldots\ \mathtt{,}b_n\mathtt{)}$ models parameter passing in the same way as in context-insensitive analysis, but it now updates the call context and takes unreachable into account:

$$[\![v]\!](c) = \bigsqcup_{\substack{w \in pred(v) \wedge \\ c = w \wedge \\ c' \in Contexts \wedge \\ [\![w]\!](c') \neq \mathsf{unreachable}}} s_w^{c'}$$

where

$$s_w^{c'} = \bot[b_1 \mapsto eval([\![w]\!](c'), E_1^w), \ldots, b_n \mapsto eval([\![w]\!](c'), E_n^w)]$$

Compared to the context-insensitive variant, the abstract state at v is now parameterized by the context c, and we only include information from the call nodes that match c. In this simple case where $k = 1$ there is no direct connection between c and c' (the context at the call node), but for larger values of k it is necessary to express how the call site is pushed onto the stack (represented by the call string).

Exercise 7.8: Verify that this constraint rule for function entry nodes indeed leads to the lattice element shown above for the example program.

Expressed using inequations instead, the constraint rule for a function entry node v where $w \in pred(v)$ is a caller and $c' \in Contexts$ is a call context can be

written as follows, which may be more intuitively clear.

$$[\![v]\!](w) \sqsupseteq s_w^{c'} \quad \text{if } [\![w]\!](c') \neq \mathsf{unreachable}$$

Informally, for any call context c' at the call node w, an abstract state $s_w^{c'}$ is built by evaluating the function arguments and propagated to call context w at the function entry node v.

Exercise 7.9: Give a constraint rule for the entry node of the special function `main`. (Remember that `main` is always reachable in context ϵ and that the values of its parameters can be any integers.)

The constraint rule for an after-call node v with associated call node v' and function exit node $w \in pred(v)$ merges the abstract state from the call node and the return value from the exit node, now taking the call contexts into account:

$$[\![v]\!](c) = [\![v']\!](c)[X \mapsto [\![w]\!](v')(\texttt{result})]$$
$$\text{if } [\![v']\!](c) \neq \mathsf{unreachable} \ \wedge \ [\![w]\!](v') \neq \mathsf{unreachable}$$

Notice that v' is both a call node and a call context, and the abstract value of `result` is obtained from the exit node w in call context v'.

Exercise 7.10: Write and solve the constraints that are generated by the interprocedural sign analysis for the program from Exercise 7.4, this time with context sensitivity using the call string approach with $k = 1$. (Even though this program does not need context sensitivity to be analyzed precisely, it illustrates the mechanism behind the call string approach.)

Exercise 7.11: Assume we have analyzed a program P with $Callers = \{c_1, c_2\}$ using the interprocedural sign analysis with call-string context sensitivity with $k = 2$, and the analysis result contains the following lattice element for the exit node of a function named `foo`:

$[\epsilon \mapsto \mathsf{unreachable},$
$(c_1) \mapsto \mathsf{unreachable},$
$(c_2) \mapsto \mathsf{unreachable},$
$(c_1, c_1) \mapsto [\texttt{result} \mapsto \texttt{-}],$
$(c_2, c_1) \mapsto \mathsf{unreachable},$
$(c_1, c_2) \mapsto [\texttt{result} \mapsto \texttt{+}],$
$(c_2, c_2) \mapsto \mathsf{unreachable}]$

Explain informally what this tells us about the program P.

Exercise 7.12: Write a TIP program that needs the call string bound $k = 2$ or higher to be analyzed with optimal precision using the sign analysis. That is, some variable in the program is assigned the abstract value `?` by the analysis if and only if $k < 2$.

Exercise 7.13: Generalize the constraint rules shown above to work with any $k \geq 1$, not just $k = 1$.

In summary, the call string approach distinguishes calls to the same function based on the call sites that appear in the call stack. In practice, $k = 1$ sometimes gives inadequate precision, and $k \geq 2$ is generally too expensive. For this reason, a useful strategy is to select k individually for each call site, based on heuristics.

7.4 Context Sensitivity with the Functional Approach

Consider this variant of the program from Section 7.2:

```
f(z) {
  return z*42;
}

main() {
  var x,y;
  x = f(42);  // call 1
  y = f(87);  // call 2
  return x + y;
}
```

The call string approach with $k \geq 1$ will analyze the `f` function twice, which is unnecessary because the abstract value of the argument is + at both calls. Rather than distingushing calls based on information about control flow from the call stack, the *functional approach* to context sensitivity distinguishes calls based on the data from the abstract states at the calls. In the most general form, the functional approach uses

$$Contexts = States$$

although a subset often suffices. With this set of call contexts, the analysis lattice becomes

$$\big(States \to \mathit{lift}(States)\big)^n$$

which clearly leads to a significant increase of the theoretical worst-case complexity compared to context insensitive analysis.

Exercise 7.14: What is the height of this lattice, expressed in terms of $h = height(States)$ and $s = |States|$?

The idea is that a lattice element for a CFG node v is a map $m : States \to \mathit{lift}(States)$ such that $m(s)$ approximates the possible states at v given that the current function containing v was entered in a state that matches s. The situation $m(s) = \mathsf{unreachable}$ means that there is no execution of the program where the

function is entered in a state that matches s and v is reached. If v is the exit node of a function f, the map m is a *summary* of f, mapping abstract entry states to abstract exit states, much like a transfer function (see Section 5.10) models the effect of executing a single instruction but now for an entire function.

Returning to the example program from Section 7.2 (page 81), we will now define the analysis constraints such that, in particular, at the exit of the function `f`, we obtain the lattice element[3]

$$\big[\bot[\mathtt{z} \mapsto \mathtt{0}] \mapsto \bot[\mathtt{z} \mapsto \mathtt{0}, \mathtt{result} \mapsto \mathtt{0}],\\ \bot[\mathtt{z} \mapsto +] \mapsto \bot[\mathtt{z} \mapsto +, \mathtt{result} \mapsto +],\\ \textit{all other contexts} \mapsto \mathsf{unreachable}\big]$$

which shows that the exit of `f` is unreachable unless `z` is `0` or + at the entry of the function, and that the sign of `result` at the exit is the same as the sign of `z` at the input. In particular, the element - maps to unreachable because `f` is never called with negative inputs in the program.

The constraint rule for an entry node v of a function $f(b_1, \ldots, b_n)$ is similar to the call strings approach, except that the context is updated differently:

$$[\![v]\!](c) = \bigsqcup_{\substack{w \in \mathit{pred}(v) \,\wedge \\ c = s_w^{c'} \,\wedge \\ c' \in \mathit{Contexts} \,\wedge \\ [\![w]\!](c') \neq \mathsf{unreachable}}} s_w^{c'}$$

(The abstract state $s_w^{c'}$ is defined as in Section 7.3.) In this constraint rule, the abstract state computed for the call context c at the entry node v only includes information from the calls that produce an entry state $s_w^{c'}$ if $c = s_w^{c'}$, independent of the context c' at the call node w.

Exercise 7.15: Verify that this constraint rule for function entry nodes indeed leads to the lattice element shown above for the example program.

Expressed using inequations, the constraint rule can be written as follows, where v is a function entry node, $w \in \mathit{pred}(v)$ is a call to the function, and $c' \in \mathit{Contexts}$:

$$[\![v]\!](s_w^{c'}) \sqsupseteq s_w^{c'}$$

This rule shows that at the call w in context c', the abstract state $s_w^{c'}$ is propagated to the function entry node v in a context that is identical to $s_w^{c'}$.

Exercise 7.16: Give a constraint rule for the entry node of the special function `main`. (Remember that `main` is always reachable and that the values of its parameters can be any integers.)

[3] We here use the map update notation described on page 33 and the fact that the bottom element of a map lattice maps all inputs to the bottom element of the codomain, so $\bot[\mathtt{z} \mapsto \mathtt{0}]$ denotes the function that maps all variables to $\bot$, except `z` which is mapped to `0`.

The constraint rule for an after-call node v with associated call node v' and function exit node $w \in pred(v)$ merges the abstract state from the call node and the return value from the exit node, while taking the call contexts into account:

$$[\![v]\!](c) = [\![v']\!](c)[X \mapsto [\![w]\!](s^c_{v'})(\texttt{result})]$$
$$\text{if } [\![v']\!](c) \neq \mathsf{unreachable} \ \wedge \ [\![w]\!](s^c_{v'}) \neq \mathsf{unreachable}$$

To find the relevant context for the function exit node, this rule builds the same abstract state as the one built at the call node.

Exercise 7.17: Assume we have analyzed a program P using context sensitive interprocedural sign analysis with the functional approach, and the analysis result contains the following lattice element for the exit node of a function named `foo`:

$$\big[[\texttt{x} \mapsto \texttt{-}, \texttt{y} \mapsto \texttt{-}, \texttt{result} \mapsto \bot] \mapsto [\texttt{x} \mapsto \texttt{+}, \texttt{y} \mapsto \texttt{+}, \texttt{result} \mapsto \texttt{+}],$$
$$[\texttt{x} \mapsto \texttt{+}, \texttt{y} \mapsto \texttt{+}, \texttt{result} \mapsto \bot] \mapsto [\texttt{x} \mapsto \texttt{-}, \texttt{y} \mapsto \texttt{-}, \texttt{result} \mapsto \texttt{-}],$$
$$\textit{all other contexts} \mapsto \mathsf{unreachable}\big]$$

Explain informally what this tells us about the program P. What could `foo` look like?

Exercise 7.18: Write and solve the constraints that are generated by the interprocedural sign analysis for the program from Exercise 7.4, this time with context sensitivity using the functional approach.

Context sensitivity with the functional approach as presented here gives optimal precision, in the sense that it is as precise as if inlining all function calls (even recursive ones). This means that it completely avoids the problem with dataflow along interprocedurally invalid paths.

Exercise 7.19: Show that this claim about the precision of the functional approach is correct.

Due to the high worst-case complexity, in practice the functional approach is often applied selectively, either only on some functions or using call contexts that only consider some of the program variables. One choice is *parameter sensitivity* where the call contexts are defined by the abstract values of the function parameters but not other parts of the program state. In the version of TIP used in this chapter, there are no pointers or global variables, so the entire program state at function entries is defined by the values of the parameters, which means that the analysis presented in this section coincides with parameter sensitivity. When analyzing object oriented programs, a popular choice is *object sensitivity*, which is essentially a variant of the functional approach that distinguishes calls not on the entire abstract states at function entries but only on the abstract values of the receiver objects.

Chapter 8

Control Flow Analysis

If we introduce higher-order functions, objects, or function pointers into the programming language, then control flow and dataflow suddenly become intertwined. At each call site, it is no longer trivial to see which code is being called. The task of *control flow analysis* is to conservatively approximate the interprocedural control flow, also called the *call graph*, for such programs.

8.1 Closure Analysis for the λ-calculus

Control flow analysis in its purest form can best be illustrated by the classical λ-calculus:

$$\begin{array}{rl} E \to & \lambda X.E \\ \mid & X \\ \mid & E\ E \end{array}$$

(In Section 8.3 we demonstrate this analysis technique on the TIP language.) For simplicity we assume that all λ-bound variables are distinct. To construct a CFG for a term in this calculus, we need to approximate for every expression E the set of *closures* to which it may evaluate. A closure can be modeled by a symbol of the form λX that identifies a concrete λ-abstraction. This problem, called *closure analysis*, can be solved using the techniques from Chapters 4 and 5. However, since the intraprocedural control flow is trivial in this language, we might as well perform the analysis directly on the AST.

The lattice we use is the powerset of closures occurring in the given term ordered by subset inclusion. For every AST node v we introduce a constraint variable $[\![v]\!]$ denoting the set of resulting closures. For an abstraction $\lambda X.E$ we have the constraint

$$\lambda X \in [\![\lambda X.E]\!]$$

(the function may certainly evaluate to itself), and for an application $E_1 E_2$ we have the *conditional* constraint

$$\lambda X \in [\![E_1]\!] \Rightarrow \left([\![E_2]\!] \subseteq [\![X]\!] \ \wedge \ [\![E]\!] \subseteq [\![E_1 E_2]\!]\right)$$

for every closure $\lambda X.E$, which models that the actual argument may flow into the formal argument and that the value of the function body is among the possible results of the function call.

Exercise 8.1: Show how the resulting constraints can be expressed as monotone constraints and solved by a fixed-point computation.

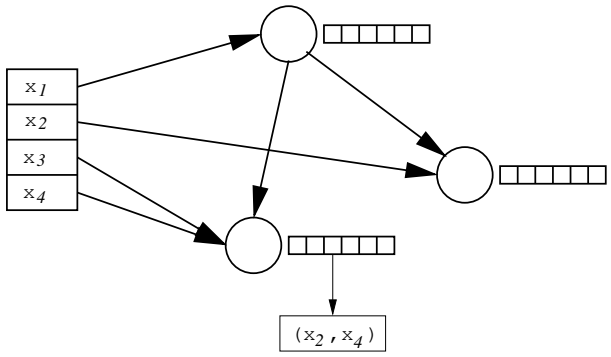
8.2 The Cubic Algorithm

The constraints for closure analysis are an instance of a general class that can be solved in cubic time. Many problems fall into this category, so we will investigate the algorithm more closely.

We have a finite set of *tokens* $\{t_1, \ldots, t_k\}$ and a finite set of *variables* $x_1, \ldots, x_n$ whose values are sets of tokens. Our task is to read a collection of *constraints* of the form $t \in x$ or $t \in x \Rightarrow y \subseteq z$ and produce the minimal solution.

Exercise 8.2: Show that a unique minimal solution exists, since solutions are closed under intersection.

The algorithm is based on a simple data structure. Each variable is mapped to a node in a directed acyclic graph (DAG). Each node has an associated bitvector belonging to $\{0,1\}^k$, initially defined to be all 0's. Each bit has an associated list of pairs of variables, which is used to model conditional constraints. The edges in the DAG reflect inclusion constraints. An example graph may look like:



Constraints are added one at a time, and the bitvectors will at all times directly represent the minimal solution of the constraints seen so far.

A constraint of the form $t \in x$ is handled by looking up the node associated with x and setting the corresponding bit to 1. If its list of pairs was not empty,

then an edge between the nodes corresponding to y and z is added for every pair (y, z) and the list is emptied. A constraint of the form $t \in x \Rightarrow y \subseteq z$ is handled by first testing if the bit corresponding to t in the node corresponding to x has value 1. If this is so, then an edge between the nodes corresponding to y and z is added. Otherwise, the pair (y, z) is added to the list for that bit.

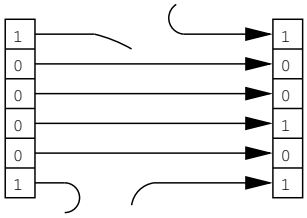
If a newly added edge forms a cycle, then all nodes on that cycle can be merged into a single node, which implies that their bitvectors are unioned together and their pair lists are concatenated. The map from variables to nodes is updated accordingly. In any case, to reestablish all inclusion relations we must propagate the values of each newly set bit along all edges in the graph.

To analyze the time complexity this algorithm, we assume that the numbers of tokens and variables are both $\mathcal{O}(n)$. This is clearly the case in closure analysis of a program of size n.

Merging DAG nodes on cycles can be done at most $\mathcal{O}(n)$ times. Each merger involves at most $\mathcal{O}(n)$ nodes and the union of their bitvectors is computed in time at most $\mathcal{O}(n^2)$. The total for this part is $\mathcal{O}(n^3)$.

New edges are inserted at most $\mathcal{O}(n^2)$ times. Constant sets are included at most $\mathcal{O}(n^2)$ times, once for each $t \in x$ constraint.

Finally, to limit the cost of propagating bits along edges, we imagine that each pair of corresponding bits along an edge are connected by a tiny bitwire. Whenever the source bit is set to 1, that value is propagated along the bitwire which then is broken:



Since we have at most n^3 bitwires, the total cost for propagation is $\mathcal{O}(n^3)$. Adding up, the total cost for the algorithm is also $\mathcal{O}(n^3)$. The fact that this seems like a lower bound as well is referred to as the *cubic time bottleneck*.

The kinds of constraints covered by this algorithm is a simple case of the more general *set constraints*, which allows richer constraints on sets of finite terms. General set constraints are also solvable but in time $\mathcal{O}(2^{2^n})$.

8.3 TIP with Function Pointers

Consider now our tiny language TIP where we allow functions pointers. For a computed function call

$$E \to (E)(E_1, \ldots, E_n)$$

we cannot see from the syntax which functions may be called. A coarse but sound CFG could be obtained by assuming that *any* function with the right

number of arguments could be called. However, we can do much better by performing a control flow analysis. Note that a direct function call $f(E_1,\ldots,E_n)$ may be seen as syntactic sugar for the general notation $(f)(E_1,\ldots,E_n)$.

Our lattice is the powerset of the set of tokens containing X for every function name X, ordered by subset inclusion. For every syntax tree node v we introduce a constraint variable $[\![v]\!]$ denoting the set of functions v could point to. For a function named f we have the constraint

$$f \in [\![f]\!]$$

for assignments X=E we have the constraint

$$[\![E]\!] \subseteq [\![X]\!]$$

and, finally, for computed function calls $(E)(E_1,\ldots,E_n)$ we have for every definition of a function f with arguments $a_f^1,\ldots,a_f^n$ and return expression E'_f this constraint:

$$f \in [\![E]\!] \;\Rightarrow\; \left([\![E_1]\!] \subseteq [\![a_f^1]\!] \;\wedge\; \cdots \;\wedge\; [\![E_n]\!] \subseteq [\![a_f^n]\!] \;\wedge\; [\![E'_f]\!] \subseteq [\![(E)(E_1,\ldots,E_n)]\!]\right)$$

A still more precise analysis could be obtained if we restrict ourselves to typable programs and only generate constraints for those functions f for which the call would be type correct.

Given this inferred information, we can construct a CFG as before but with edges between a call site and all possible target functions according to the control flow analysis. Consider the following example program:

```
inc(i) { return i+1; }
dec(j) { return j-1; }
ide(k) { return k; }

foo(n,f) {
  var r;
  if (n==0) { f=ide; }
  r = (f)(n);
  return r;
}

main() {
  var x,y;
  x = input;
  if (x>0) { y = foo(x,inc); } else { y = foo(x,dec); }
  return y;
}
```

The control flow analysis generates the following constraints:

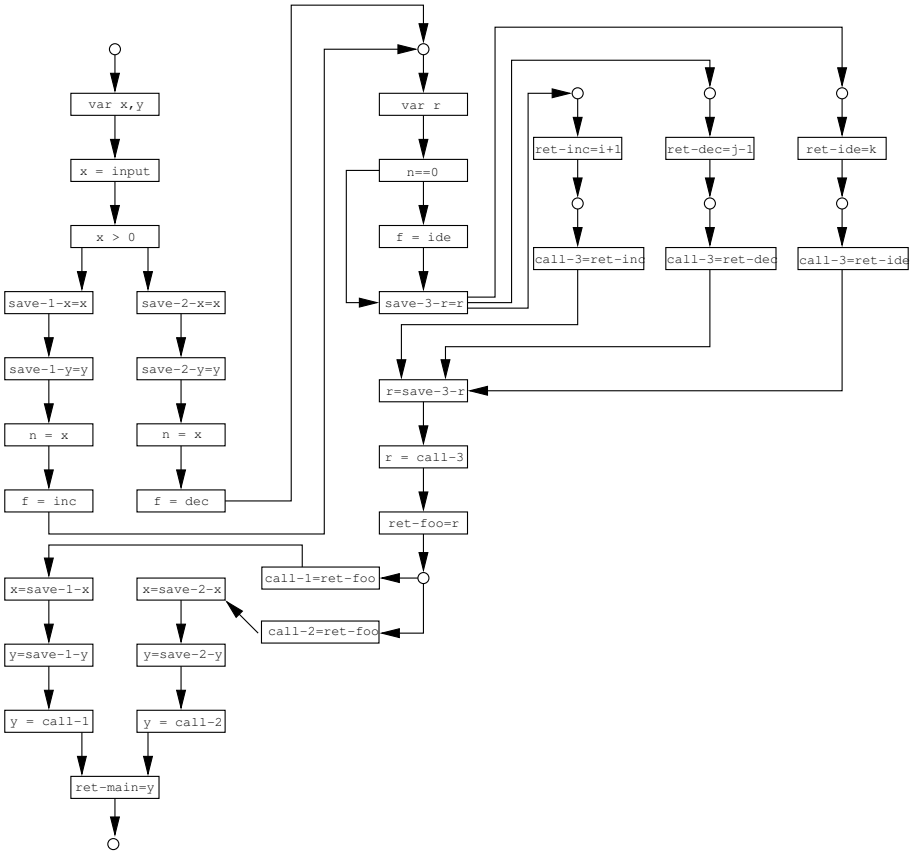
$\texttt{inc} \in [\![\texttt{inc}]\!]$

$$
\begin{array}{l}
\texttt{dec} \in [\![\texttt{dec}]\!] \\
\texttt{ide} \in [\![\texttt{ide}]\!] \\
[\![\texttt{ide}]\!] \subseteq [\![\texttt{f}]\!] \\
[\![\texttt{(f)(n)}]\!] \subseteq [\![\texttt{r}]\!] \\
\texttt{inc} \in [\![\texttt{f}]\!] \Rightarrow [\![\texttt{n}]\!] \subseteq [\![\texttt{i}]\!] \ \wedge\ [\![\texttt{i+1}]\!] \subseteq [\![\texttt{(f)(n)}]\!] \\
\texttt{dec} \in [\![\texttt{f}]\!] \Rightarrow [\![\texttt{n}]\!] \subseteq [\![\texttt{j}]\!] \ \wedge\ [\![\texttt{j-1}]\!] \subseteq [\![\texttt{(f)(n)}]\!] \\
\texttt{ide} \in [\![\texttt{f}]\!] \Rightarrow [\![\texttt{n}]\!] \subseteq [\![\texttt{k}]\!] \ \wedge\ [\![\texttt{k}]\!] \subseteq [\![\texttt{(f)(n)}]\!] \\
[\![\texttt{input}]\!] \subseteq [\![\texttt{x}]\!] \\
[\![\texttt{foo(x,inc)}]\!] \subseteq [\![\texttt{y}]\!] \\
[\![\texttt{foo(x,dec)}]\!] \subseteq [\![\texttt{y}]\!] \\
\texttt{foo} \in [\![\texttt{foo}]\!] \\
\texttt{foo} \in [\![\texttt{foo}]\!] \Rightarrow [\![\texttt{x}]\!] \subseteq [\![\texttt{n}]\!] \ \wedge\ [\![\texttt{inc}]\!] \subseteq [\![\texttt{f}]\!] \ \wedge\ [\![\texttt{r}]\!] \subseteq [\![\texttt{foo(x,inc)}]\!] \\
\texttt{foo} \in [\![\texttt{foo}]\!] \Rightarrow [\![\texttt{x}]\!] \subseteq [\![\texttt{n}]\!] \ \wedge\ [\![\texttt{dec}]\!] \subseteq [\![\texttt{f}]\!] \ \wedge\ [\![\texttt{r}]\!] \subseteq [\![\texttt{foo(x,dec)}]\!]
\end{array}
$$

The nonempty values of the least solution are:

$$
\begin{array}{l}
[\![\texttt{inc}]\!] = \{\texttt{inc}\} \\
[\![\texttt{dec}]\!] = \{\texttt{dec}\} \\
[\![\texttt{ide}]\!] = \{\texttt{ide}\} \\
[\![\texttt{f}]\!] = \{\texttt{inc}, \texttt{dec}, \texttt{ide}\} \\
[\![\texttt{foo}]\!] = \{\texttt{foo}\}
\end{array}
$$

On this basis, we can construct the following monovariant interprocedural CFG for the program:



which then can be used as basis for subsequent interprocedural dataflow analyses.

Exercise 8.3: Consider the following TIP program:

```
f(y,z) {
  return (y)(z);
}
g(x) {
  return x+1;
}
main(a) {
  return f(g,f(g,a));
}
```

(a) Write the constraints that are produced by the control-flow analysis for this program.

(b) Are the constraints solvable? If so, write the solution. If not, explain why there is no solution.

Exercise 8.4: As an alternative to the analysis described above, we can use dataflow analysis in the style of Chapter 5. Design a dataflow analysis that performs control flow analysis for TIP. (Hint: choose a suitable lattice and define appropriate dataflow constraints.) Then explain briefly how your analysis handles the following TIP program, compared to using the flow-insensitive analysis described above.

```
inc(x) {
  return x+1;
}
dec(y) {
  return y-1;
}
main(a) {
  var t;
  t = inc;
  a = (t)(a);
  t = dec;
  a = (t)(a);
  return a;
}
```

8.4 Control Flow in Object Oriented Languages

A language with function pointers or higher-order functions must use the kind of control flow analysis described in the previous sections to obtain a reasonably precise CFG. For common object-oriented languages, such as Java or C#, it is also useful, but the added structure provided by the class hierarchy and the type system permits some simpler alternatives. In the object-oriented setting the question is which method implementations may be executed at a given method invocation site:

```
x.m(a,b,c)
```

The simplest solution is to scan the class library and select any method named `m` whose signature accepts the types of the actual arguments. A better choice, called *Class Hierarchy Analysis (CHA)*, is to consider only the part of the class hierarchy that is spanned by the declared type of `x`. A further refinement, called *Rapid Type Analysis (RTA)*, is to restrict further to the classes of which objects are actually allocated. Yet another technique, called *Variable Type Analysis (VTA)*, performs *intraprocedural* control flow analysis while making conservative assumptions about the remaining program.

These techniques are of course much faster than full-blown control flow analysis, and for real-life programs they are often sufficiently precise.

Chapter 9

Pointer Analysis

The final extension of the TIP language introduces simple pointers and dynamic memory allocation. Since our toy version of `alloc` only allocates a single cell, we cannot build arbitrary structures in the heap. However, the main problems with pointers are amply represented in the language fragment that we consider.

To illustrate the problem with pointers, assume we with to perform a sign analysis of TIP code like this:

```
...
*x = 42;
*y = -87;
z = *z;
```

Here, the value of `z` depends on whether or not `x` and `y` are aliases, meaning that they point to the same cell. Without knowledge of such aliasing information, it quickly becomes impossible to produce useful dataflow and control-flow analysis results.

9.1 Allocation-Site Abstraction

We first focus on intraprocedural analysis and postpone treatment of function calls to Section 9.4.

The most important information that must be obtained is the set of possible memory cells that the pointers may point to. There are of course arbitrarily many possible cells during execution, so we must select some finite abstraction. A common choice, called *allocation-site abstraction*, is to introduce an abstract cell X for every program variable named X and an abstract cell `alloc-`i, where i is a unique index, for each occurrence of an `alloc` operation in the program. Each abstract cell represents the set of cells at runtime that are allocated at the same source location, hence the name allocation-site abstraction. We use Cells to

denote the set of abstract cells for the given program, and we use $Locs$ to denote the set of abstract locations of the cells, written &$c \in Locs$ for every $c \in Cells$.

The first analyses that we shall study are flow-insensitive. The end result of such an analysis is a function $pt : Vars \to 2^{Cells}$ that for each pointer variable X returns the set $pt(X)$ of cells it may point to. We wish to perform a conservative analysis that computes sets that may be too large but never too small.

Given such *points-to* information, many other facts can be approximated. If we wish to know whether pointer variables `x` and `y` *may* be aliases, then a safe answer is obtained by checking whether $pt(\texttt{x}) \cap pt(\texttt{y})$ is nonempty.

The initial values of variables and heap cells are undefined in TIP programs, however, for these flow-insensitive points-to analyses we assume that all variables and heap cells are initialized before they are used. (In other words, these analyses are sound only for programs that never read from uninitialized variables or heap cells.)

An almost-trivial analysis, called *address taken*, is to simply return all possible abstract cells, except that X is only included if the expression &X occurs in the given program. This only suffices for very simple applications, so more ambitious approaches are usually preferred. If we restrict ourselves to typable programs, then any points-to analysis could be improved by removing those locations whose types are not equal to that of the pointer variable.

9.2 Andersen's Algorithm

One approach to points-to analysis is quite similar to control flow analysis. For each cell c we introduce a constraint variable $[\![c]\!]$ ranging over sets of locations.

The analysis assumes that the program has been normalized so that every pointer operation is of one of these six kinds:

- X `= alloc`
- X_1 `= &`X_2
- X_1 `=` X_2
- X_1 `= *`X_2
- `*`X_1 `=` X_2
- X `= null`

Exercise 9.1: Show how this normalization can be performed systematically by introducing fresh temporary variables.

Exercise 9.2: Normalize the single statement `**x = **y`.

For each of these pointer operations we then generate constraints:

X = alloc:	&alloc-i $\in [\![X]\!]$
X_1 = &X_2:	&$X_2 \in [\![X_1]\!]$
X_1 = X_2:	$[\![X_2]\!] \subseteq [\![X_1]\!]$
X_1 = *X_2:	&$\alpha \in [\![X_2]\!] \Rightarrow [\![\alpha]\!] \subseteq [\![X_1]\!]$
*X_1 = X_2:	&$\alpha \in [\![X_1]\!] \Rightarrow [\![X_2]\!] \subseteq [\![\alpha]\!]$

The `null` assignment is ignored, since it corresponds to the trivial constraint $\emptyset \subseteq [\![X]\!]$. Notice that these constraints match the requirements of the cubic algorithm from Section 8.2. The resulting points-to function is defined as:

$$pt(p) = \{\alpha \in Cells \mid \&\alpha \in [\![p]\!]\}$$

Consider the following example program fragment.

```
p = alloc;
x = y;
x = z;
*p = z;
p = q;
q = &y;
x = *p;
p = &z;
```

Andersen's algorithm generates these constraints:

&alloc-1 $\in [\![$p$]\!]$
$[\![$y$]\!] \subseteq [\![$x$]\!]$
$[\![$z$]\!] \subseteq [\![$x$]\!]$
&$\alpha \in [\![$p$]\!] \Rightarrow [\![$z$]\!] \subseteq [\![\alpha]\!]$
$[\![$q$]\!] \subseteq [\![$p$]\!]$
&y $\in [\![$q$]\!]$
&$\alpha \in [\![$p$]\!] \Rightarrow [\![\alpha]\!] \subseteq [\![x]\!]$
&z $\in [\![$p$]\!]$

The least solution is quite precise (here showing only the nonempty values):

$pt($p$) = \{$alloc-1, y, z$\}$
$pt($q$) = \{$y$\}$

Note that while this algorithm is flow insensitive, the directionality of the constraints implies that the dataflow is still modeled with some accuracy.

Exercise 9.3: Use Andersen's algorithm to compute the points-to sets for the variables in the following program fragment:

```
a = &d;
b = &e;
a = b;
*a = alloc;
```

Exercise 9.4: Use Andersen's algorithm to compute the points-to sets for the variables in the following program fragment:

```
z = &x;
w = &a;
a = 42;
if (a > b) {
  *z = &a;
  y = &b;
} else {
  x = &b;
  y = w;
}
```

9.3 Steensgaard's Algorithm

An interesting alternative is Steensgaard's algorithm, which performs a coarser analysis essentially by viewing assignments as being bidirectional. The analysis can be expressed elegantly using term unification. We use a term variable $[\![c]\!]$ for every cell c and a term constructor $\&t$ representing a pointer to t. (Notice the change in notation compared to Section 9.2: here, $[\![c]\!]$ is a term variable and does not directly denote a set of abstract locations.)

$$\begin{array}{rl}
X \texttt{ = alloc}: & [\![X]\!] = \&[\![\texttt{alloc-}i]\!] \\
X_1 \texttt{ = } \&X_2: & [\![X_1]\!] = \&[\![X_2]\!] \\
X_1 \texttt{ = } X_2: & [\![X_1]\!] = [\![X_2]\!] \\
X_1 \texttt{ = } *X_2: & [\![X_2]\!] = \&\alpha \ \wedge \ [\![X_1]\!] = \alpha \\
*X_1 \texttt{ = } X_2: & [\![X_1]\!] = \&\alpha \ \wedge \ [\![X_2]\!] = \alpha
\end{array}$$

Each α denotes a fresh term variable.

As usual, term constructors satisfy the general term equality axiom:

$$\&\alpha_1 = \&\alpha_2 \ \Rightarrow \ \alpha_1 = \alpha_2$$

The resulting points-to function is defined as:

$$\texttt{pt}(p) = \{t \in \mathit{Cells} \mid [\![p]\!] = \&[\![t]\!]\}$$

For the example program from Section 9.2, Steensgaard's algorithm generates the following constraints:

$[\![\texttt{p}]\!] = \&[\![\texttt{alloc-1}]\!]$
$[\![\texttt{x}]\!] = [\![\texttt{y}]\!]$
$[\![\texttt{x}]\!] = [\![\texttt{z}]\!]$
$[\![\texttt{p}]\!] = \&\alpha_1 \qquad [\![\texttt{z}]\!] = \alpha_1$

$$[\![\texttt{p}]\!] = [\![\texttt{q}]\!]$$
$$[\![\texttt{q}]\!] = \&[\![\texttt{y}]\!]$$
$$[\![\texttt{x}]\!] = \alpha_2 \qquad [\![\texttt{p}]\!] = \&\alpha_2$$
$$[\![\texttt{p}]\!] = \&[\![\texttt{z}]\!]$$

This in turn implies that

$$pt(\texttt{p}) = pt(\texttt{q}) = \{\texttt{alloc-1}, \texttt{y}, \texttt{z}\}$$

which is less precise than Andersen's algorithm, but using the faster algorithm.

Exercise 9.5: Use Steensgaard's algorithm to compute the points-to sets for the two programs from Exercise 9.3 and Exercise 9.4.

Exercise 9.6: Can the constraint rule for $X_1 = *X_2$ be simplified from

$$[\![X_2]\!] = \&\alpha \ \wedge \ [\![X_1]\!] = \alpha$$

to

$$[\![X_2]\!] = \&[\![X_1]\!]$$

without affecting the analysis results?

Similarly, can the constraint rule for $*X_1 = X_2$ be simplified from

$$[\![X_1]\!] = \&\alpha \ \wedge \ [\![X_2]\!] = \alpha$$

to

$$[\![X_1]\!] = \&[\![X_2]\!]$$

without affecting the analysis results?

9.4 Interprocedural Points-To Analysis

In languages with both function pointers and heap pointers, function pointers may be stored in the heap, which makes it difficult to perform control flow analysis before points-to analysis. But it is also difficult to perform interprocedural points-to analysis without the information from a control flow analysis. For example, the following function call uses a function pointer accessed via a heap pointer dereference and also passes a pointer as argument:

```
(*x)(x);
```

The solution to this chicken-and-egg problem is to perform control flow analysis and points-to analysis *simultaneously*.

To express the combined algorithm, we assume that all function calls are normalized to the form

X = `(`X'`)(`X_1`,`$\ldots$`,`X_n`);`

so that the involved expressions are all variables. Similarly, all return expressions are assumed to be just variables.

Exercise 9.7: Show how to perform such normalization in a systematic manner.

Andersen's algorithm is already similar to control flow analysis, and it can simply be extended with the appropriate constraints. A reference to a constant function f generates the constraint:

$$f \in [\![f]\!]$$

The computed function call generates the constraint

$$f \in [\![X']\!] \Rightarrow \left([\![X_1]\!] \subseteq [\![X'_1]\!] \ \wedge \ \cdots \ \wedge \ [\![X_n]\!] \subseteq [\![X'_n]\!] \ \wedge \ [\![X'']\!] \subseteq [\![X]\!]\right)$$

for every occurrence of a function definition with n parameters

f`(`X'_1`,`$\ldots$`,`X'_n`) { `$\ldots$`return` X''`; }`

This will maintain the precision of the control flow analysis.

Exercise 9.8: Design a *context-sensitive* variant of the Andersen-style points-to analysis. (Hint: see Sections 7.2–7.4.)

Exercise 9.9: Continuing Exercise 9.8, can we still use the cubic algorithm (Section 8.2) to solve the analysis constraints? If so, is the analysis time still $\mathcal{O}(n^3)$ where n is the size of the program being analyzed?

9.5 Null Pointer Analysis

We are now also able to define an analysis that detects `null` dereferences. Specifically, we want to ensure that `*`X is only executed when X is not `null`. Let us consider intraprocedural analysis, so we can ignore function calls.

As before, we assume that the program is normalized, so that all pointer manipulations are of the six kinds described in Section 9.2 The basic lattice we use, called $Null$, is:

where the bottom element `NN` means *definitely not* `null` and the top element `?` represents values that may be `null`. We then form the following map lattice for abstract states:

$$States = Cells \to Null$$

For every CFG node v we introduce a constraint variable $[\![v]\!]$ denoting an element from the map lattice. We shall use each constraint variable to describe an abstract state for the program point immediately after the node.

For all nodes that do not involve pointer operations we have the constraint:

$$[\![v]\!] = \mathit{JOIN}(v)$$

where

$$\mathit{JOIN}(v) = \bigsqcup_{w \in pred(v)} [\![w]\!]$$

For a heap load operation X_1 `=` `*`X_2 we need to model the change of the program variable X_1. Our abstraction has a single abstract cell for X_1. With the assumption of intraprocedural analysis, that abstract cell represents a single concrete cell. (With an interprocedural analysis, we would need to take into account that each stack frame at runtime has an instance of the variable.) For the expression `*`X_2 we can ask the points-to analysis for the possible cells $pt(X_2)$. With these observations, we can give a constraint for heap load operations:

$$X_1 \texttt{ = *}X_2\texttt{:} \quad [\![v]\!] = \mathit{load}(\mathit{JOIN}(v), X_1, X_2)$$

where

$$\mathit{load}(\sigma, X_1, X_2) = \sigma[X_1 \mapsto \bigsqcup_{\alpha \in pt(X_2)} \sigma(\alpha)]$$

Similar reasoning gives constraints for the other operations that affect pointer variables:

$$\begin{array}{rl}
X \texttt{ = alloc}\texttt{:} & [\![v]\!] = \mathit{JOIN}(v)[X \mapsto \texttt{NN}, \texttt{alloc-}\mathit{i} \mapsto \texttt{?}] \\
X_1 \texttt{ = \&}X_2\texttt{:} & [\![v]\!] = \mathit{JOIN}(v)[X_1 \mapsto \texttt{NN}] \\
X_1 \texttt{ = } X_2\texttt{:} & [\![v]\!] = \mathit{JOIN}(v)[X_1 \mapsto \mathit{JOIN}(v)(X_2)] \\
X \texttt{ = null}\texttt{:} & [\![v]\!] = \mathit{JOIN}(v)[X \mapsto \texttt{?}]
\end{array}$$

Exercise 9.10: Explain why the above four constraints are monotone and sound.

For a heap store operation `*`X_1 `=` X_2 we need to model the change of whatever X_1 points to. That may be multiple abstract cells, namely $pt(X_1)$. Moreover, each abstract heap cell `alloc-`i may describe multiple concrete cells. In the constraint for heap store operations, we must therefore join the new abstract value into the existing one for each affected cell in $pt(X_1)$:

$$\texttt{*}X_1 \texttt{ = } X_2\texttt{:} \quad [\![v]\!] = \mathit{store}(\mathit{JOIN}(v), X_1, X_2)$$

where

$$\mathit{store}(\sigma, X_1, X_2) = \sigma\ [\alpha \mapsto \sigma(\alpha) \sqcup \sigma(X_2)\ ]_{\alpha \in pt(X_1)}$$

The situation we here see at heap store operations where we model an assignment by joining the new abstract value into the existing one is called a

weak update. In contrast, in a *strong update* the new abstract value overwrites the existing one, which we see in the null pointer analysis at all operations that modify program variables. Strong updates are obviously more precise than weak updates in general, but it may require more elaborate analysis abstractions to detect situations where strong update can be applied soundly.

After performing the null pointer analysis of a given program, a pointer dereference $*X$ at a program point v is guaranteed to be safe if

$$JOIN(v)(X) = \texttt{NN}$$

The precision of this analysis depends of course on the quality of the underlying points-to analysis.

Consider the following buggy example program:

```
p = alloc;
q = &p;
n = null;
*q = n;
*p = n;
```

Andersen's algorithm computes the following points-to sets:

$pt(\texttt{p}) = \{\texttt{alloc-1}\}$
$pt(\texttt{q}) = \{\texttt{p}\}$
$pt(\texttt{n}) = \emptyset$

Based on this information, the `null` pointer analysis generates the following constraints:

$[\![\texttt{p=alloc}]\!] = \bot[\texttt{p} \mapsto \texttt{NN}, \texttt{alloc-1} \mapsto \texttt{?}]$
$[\![\texttt{q=\&p}]\!] = [\![\texttt{p=alloc}]\!][\texttt{q} \mapsto \texttt{NN}]$
$[\![\texttt{n=null}]\!] = [\![\texttt{q=\&p}]\!][\texttt{n} \mapsto \texttt{?}]$
$[\![\texttt{*q=n}]\!] = [\![\texttt{n=null}]\!][\texttt{p} \mapsto [\![\texttt{n=null}]\!](\texttt{p}) \sqcup [\![\texttt{n=null}]\!](\texttt{n})]$
$[\![\texttt{*p=n}]\!] = [\![\texttt{*q=n}]\!][\texttt{alloc-1} \mapsto [\![\texttt{*q=n}]\!](\texttt{alloc-1}) \sqcup [\![\texttt{*q=n}]\!](\texttt{n})]$

The least solution is:

$[\![\texttt{p=alloc}]\!] = [\texttt{p} \mapsto \texttt{NN}, \texttt{q} \mapsto \texttt{NN}, \texttt{n} \mapsto \texttt{NN}, \texttt{alloc-1} \mapsto \texttt{?}]$
$[\![\texttt{q=\&p}]\!] = [\texttt{p} \mapsto \texttt{NN}, \texttt{q} \mapsto \texttt{NN}, \texttt{n} \mapsto \texttt{NN}, \texttt{alloc-1} \mapsto \texttt{?}]$
$[\![\texttt{n=null}]\!] = [\texttt{p} \mapsto \texttt{NN}, \texttt{q} \mapsto \texttt{NN}, \texttt{n} \mapsto \texttt{?}, \texttt{alloc-1} \mapsto \texttt{?}]$
$[\![\texttt{*q=n}]\!] = [\texttt{p} \mapsto \texttt{?}, \texttt{q} \mapsto \texttt{NN}, \texttt{n} \mapsto \texttt{?}, \texttt{alloc-1} \mapsto \texttt{?}]$
$[\![\texttt{*p=n}]\!] = [\texttt{p} \mapsto \texttt{?}, \texttt{q} \mapsto \texttt{NN}, \texttt{n} \mapsto \texttt{?}, \texttt{alloc-1} \mapsto \texttt{?}]$

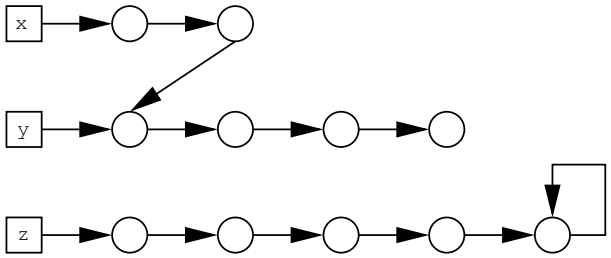
By inspecting this information, an analysis could statically detect that when `*q=n` is evaluated, which is immediately after `n=null`, the variable `q` is definitely non-`null`. On the other hand, when `*p=n` is evaluated, we cannot rule out the possibility that `p` may contain `null`.

Exercise 9.11: Show an alternative constraint for heap load operations using weak update, together with an example program where the modified analysis then gives a result that is less precise than the analysis presented above.

Exercise 9.12: Show an (unsound) alternative constraint for heap store operations using strong update, together with an example program where the modified analysis then gives a wrong result.

9.6 Flow-Sensitive Points-To Analysis

Note that we can produce interesting heaps with TIP programs, even though the `alloc` operation only allocates a single heap cell. An example of a nontrivial heap is



where `x`, `y`, and `z` are program variables. We will seek to answer questions about disjointness of the structures contained in program variables. In the example above, `x` and `y` are not disjoint whereas `y` and `z` are. Such information may be useful, for example, to automatically parallelize execution in an optimizing compiler. For such analysis, flow-insensitive reasoning is sometimes too imprecise. However, we can create a flow-sensitive variant of Andersen's analysis.

We use a lattice of *points-to graphs*, which are directed graphs in which the nodes are the abstract cells for the given program and the edges correspond to possible pointers. Points-to graphs are ordered by inclusion of their sets of edges. Thus, $\bot$ is the graph without edges and $\top$ is the completely connected graph. Formally, our lattice for abstract states is then

$$States = 2^{Cells \times Cells}$$

ordered by the usual subset inclusion. For every CFG node v we introduce a constraint variable $[\![v]\!]$ denoting a points-to graph that describes all possible stores at that program point. For the nodes corresponding to the various pointer manipulations we have these constraints:

$$\begin{array}{rl}
X \texttt{ = alloc}: & [\![v]\!] = JOIN(v)\downarrow X \cup \{(X, \texttt{alloc-}i)\} \\
X_1 \texttt{ = \&}X_2\texttt{:} & [\![v]\!] = JOIN(v)\downarrow X_1 \cup \{(X_1, X_2)\} \\
X_1 \texttt{ = } X_2\texttt{:} & [\![v]\!] = assign(JOIN(v), X_1, X_2) \\
X_1 \texttt{ = *}X_2\texttt{:} & [\![v]\!] = load(JOIN(v), X_1, X_2) \\
\texttt{*}X_1 \texttt{ = } X_2\texttt{:} & [\![v]\!] = store(JOIN(v), X_1, X_2) \\
X \texttt{ = null}: & [\![v]\!] = JOIN(v)\downarrow X
\end{array}$$

and for all other nodes:

$$[\![v]\!] = JOIN(v)$$

where

$$JOIN(v) = \bigcup_{w \in pred(v)} [\![w]\!]$$

$$\sigma{\downarrow}x = \{(s,t) \in \sigma \mid s \neq x\}$$

$$assign(\sigma, x, y) = \sigma{\downarrow}x \ \cup \{(x,t) \mid (y,t) \in \sigma\}$$

$$load(\sigma, x, y) = \sigma{\downarrow}x \ \cup \{(x,t) \mid (y,s) \in \sigma, (s,t) \in \sigma\}$$

$$store(\sigma, x, y) = \sigma \ \cup \{(s,t) \mid (x,s) \in \sigma, (y,t) \in \sigma\}$$

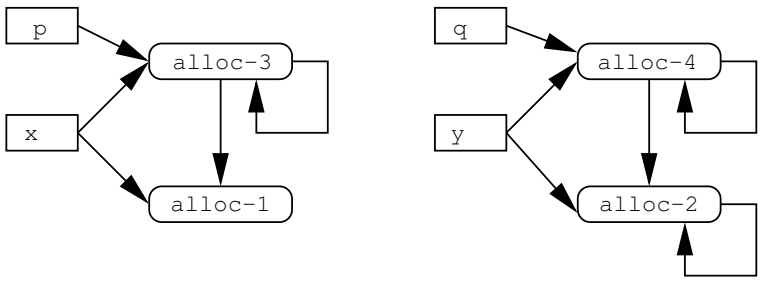
Notice that the constraint for heap store operations uses weak update.

Exercise 9.13: Explain the above constraints.

Consider now the following program:

```
var x,y,n,p,q;
x = alloc; y = alloc;
*x = null; *y = y;
n = input;
while (n>0) {
  p = alloc; q = alloc;
  *p = x; *q = y;
  x = p; y = q;
  n = n-1;
}
```

After the loop, the analysis produces the following points-to graph:



From this result we can safely conclude that `x` and `y` will always be disjoint.

Note that this analysis also computes a flow sensitive points-to map that for each program point v is defined by:

$$pt(p) = \{t \mid (p,t) \in [\![v]\!]\}$$

This analysis is more precise than Andersen's algorithm, but clearly also more expensive to perform. As an example, consider the program:

```
x = &y;
x = &z;
```

After these statements, Andersen's algorithm would predict that $pt(\mathtt{x}) = \{\mathtt{y}, \mathtt{z}\}$ whereas the flow-sensitive analysis computes $pt(\mathtt{x}) = \{\mathtt{z}\}$ for the final program point.

9.7 Escape Analysis

We earlier lamented the *escaping stack cell* error displayed by the following program, which was beyond the scope of the type analysis.

```
baz() {
  var x;
  return &x;
}

main() {
  var p;
  p=baz(); *p=1;
  return *p;
}
```

Having performed a points-to analysis, we can easily perform an *escape analysis* to catch such errors. We just need to check that the possible cells for return expressions in the points-to graph cannot reach arguments or variables defined in the function itself, since all other locations must then necessarily reside in earlier frames on the invocation stack.