Static Program Analysis

Anders Møller and Michael I. Schwartzbach

September 7, 2026

Department of Computer Science
Aarhus University, Denmark

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. For example, nobody will use an analysis designed for bug finding if it reports many false positives or if it is too slow to fit into real-world software development processes.

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, widening and narrowing, path sensitivity, relational analysis, interprocedural analysis, context sensitivity, control flow analysis, several flavors of pointer analysis, and key concepts of semantics-based abstract interpretation. 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 take 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 or type annotations) 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-time 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:

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

Chapter 1

Introduction

Static program analysis aims to automatically answer questions about the possible behaviors of programs. In this chapter, we explain why this can be useful and interesting, and we discuss the basic characteristics of analysis tools.

1.1 Applications of Static Program Analysis

Static program analysis has been used since the early 1960s in optimizing compilers. More recently, it has proven useful also for bug finding and verification tools, and in IDEs to support program development. In the following, we give some examples of the kinds of questions about program behavior that arise in these different applications.

Analysis for program optimization Optimizing compilers (including just-in-time compilers in interpreters) need to know many different properties of the program being compiled, in order to generate efficient code. A few examples of such properties are:

- Does the program contain dead code, or more specifically, is function `f` unreachable from `main`? If so, the code size can be reduced.
- Is the value of some expression inside a loop the same in every iteration? If so, the expression can be moved outside the loop to avoid redundant computations.
- Does the value of variable `x` depend on the program input? If not, it could be precomputed at compile time.
- What are the lower and upper bounds of the integer variable `x`? The answer may guide the choice of runtime representation of the variable.
- Do `p` and `q` point to disjoint data structures in memory? That may enable parallel processing.

Analysis for program correctness The most successful analysis tools that have been designed to detect errors (or verify absence of errors) target generic correctness properties that apply to most or all programs written in specific programming languages. In unsafe languages like C, such errors sometimes lead to critical security vulnerabilities. In safer languages like Java, such errors are typically less severe, but they can still cause program crashes. Examples of such properties are:

- Does there exist an input that leads to a null pointer dereference, division-by-zero, or arithmetic overflow?
- Are all variables initialized before they are read?
- Are arrays always accessed within their bounds?
- Can there be dangling references, i.e., use of pointers to memory that has been freed?
- Does the program terminate on every input? Even in reactive systems such as operating systems, the individual software components, for example device driver routines, are expected to always terminate.

Other correctness properties depend on specifications provided by the programmer for the individual programs (or libraries), for example:

- Are all assertions guaranteed to succeed? Assertions express program-specific correctness properties that are supposed to hold in all executions.
- Is function `hasNext` always called before function `next`, and is `open` always called before `read`? Many libraries have such so-called *typestate* correctness properties.
- Does the program throw an `ActivityNotFoundException` or a `SQLiteException` for some input?

With web and mobile software, information flow correctness properties have become extremely important:

- Can input values from untrusted users flow unchecked to file system operations? This would be a violation of *integrity*.
- Can secret information become publicly observable? Such situations are violations of *confidentiality*.

The increased use of concurrency (parallel or distributed computing) and event-driven execution models gives rise to more questions about program behavior:

- Are data races possible? Many errors in multi-threaded programs are caused by two threads using a shared resource without proper synchronization.
- Can the program (or parts of the program) deadlock? This is often a concern for multi-threaded programs that use locks for synchronization.

Analysis for program development Modern IDEs perform various kinds of program analysis to support debugging, refactoring, and program understanding. This involves questions such as:

- Which functions may possibly be called on line 117, or conversely, where can function `f` possibly be called from? Function inlining and other refactorings rely on such information.
- At which program points could `x` be assigned its current value? Can the value of variable `x` affect the value of variable `y`? Such questions often arise when programmers are trying to understand large codebases and during debugging when investigating why a certain bug appears.
- What types of values can variable `x` have? This kind of question often arises with programming languages where type annotations are optional or entirely absent, for example OCaml, JavaScript, or Python.

1.2 Approximative Answers

Regarding correctness, programmers routinely use testing to gain confidence that their programs work as intended, but as famously stated by Dijkstra [Dij70]: *"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 certain property.

Reasoning about the behavior of programs can be extremely difficult, even for small programs. As an example, does the following program code terminate on every integer input `n` (assuming arbitrary-precision integers)?

```
while (n > 1) {
  if (n % 2 == 0)  // if n is even, divide it by two
    n = n / 2;
  else  // if n is odd, multiply by three and add one
    n = 3 * n + 1;
}
```

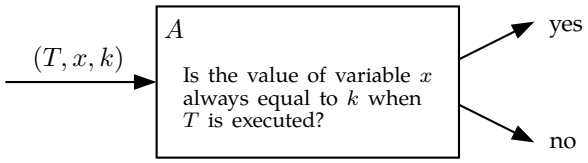
In 1937, Collatz conjectured that the answer is "yes". As of 2020, the conjecture has been checked for all inputs up to 2^{68}, but nobody has been able to prove it for all inputs [Roo19].

Even straight-line programs can be difficult to reason about. Does the following program output *true* for some integer inputs?

```
x = input; y = input; z = input;
output x*x*x + y*y*y + z*z*z == 42;
```

This was an open problem since 1954 until 2019 when the answer was found after over a million hours of computing [BS19].

Rice's theorem [Ric53] is a general result from 1953 which informally states that all interesting questions about the input/output 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, one of T's variables x, and some value k, and decides whether or not x's value is always equal to k whenever T is executed.



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 17 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 [Tur37].

At first, this seems like a discouraging result; however, this theoretical result does not prevent *approximative* answers. 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);
```

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

```
      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 (for this program, no errors are encountered 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.

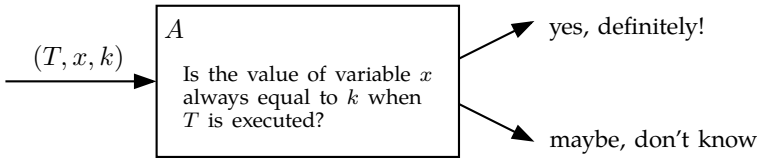
Exercise 1.1: Describe all the pointer-related errors in the above program.

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 approximations that are conservative with respect to the semantics of the programming language. The theory of semantics-based abstract interpretation presented in Chapter 12 provides a solid mathematical framework for reasoning about analysis soundness and precision. Although soundness is a laudable goal in analysis design, modern analyzers for real programming languages often cut corners by sacrificing soundness to obtain better precision and performance, for example when modeling reflection in Java [LSS+15].

1.3 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. In other words, the halting problem is not the only obstacle; approximation is inevitably necessary.

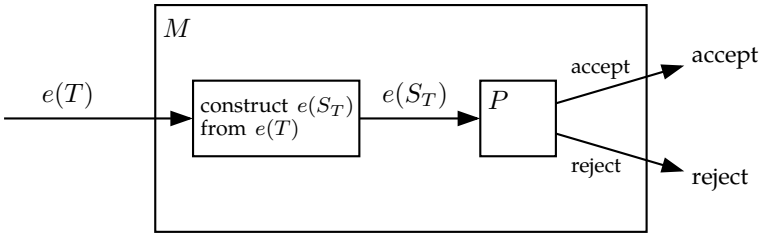
If we model programs as deterministic Turing machines, program failure can be modeled using a special *fail* state.[2] That is, on a given input, a Turing machine will eventually halt in its accept state (intuitively returning "yes"), in its reject state (intuitively returning "no"), in its fail state (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

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

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 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. By the construction of S_T, this means that T accepts input $e(T)$ within $|v|$ steps. Hence T accepts input $e(T)$, and since $T = M$, this contradicts the assumption that M rejects 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. Show how the proof can 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, expressed as a context-free grammar. 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, nested functions, objects, and type annotations. We will consider some of those features in exercises in later chapters.

Basic Expressions

The basic expressions all denote integer values:

Int → `0` | `1` | `-1` | `2` | `-2` | ...
Id → `x` | `y` | `z` | ...
Exp → *Int*
| *Id*
| *Exp* + *Exp* | *Exp* - *Exp* | *Exp* * *Exp* | *Exp* / *Exp* | *Exp* > *Exp* | *Exp* == *Exp*
| `(` *Exp* `)`

| `input`

Expressions *Exp* include integer literals *Int* and variables (identifiers) *Id*. The `input` expression reads an integer from the input stream. The comparison operators yield 0 for false and 1 for true. Function calls, pointer operations, and record operations will be added later. Throughout TIP, subexpressions are evaluated from left to right.

Statements

The simple statements *Stm* are familiar:

Stm → *Id* = *Exp* `;`
| `output` *Exp* `;`
| *Stm Stm*
|
| `if (` *Exp* `) {` *Stm* `}` [`else {` *Stm* `}`]$^{?}$
| `while (` *Exp* `) {` *Stm* `}`

We use the notation [...]$^{?}$ 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 list of parameters, local variable declarations, a body statement, and a return expression:

Fun → *Id* `(` *Id* `,` ... `,` *Id* `) {` [`var` *Id* `,` ... `,` *Id* `;`]$^{?}$ *Stm* `return` *Exp* `; }`

Function names and parameters are identifiers, like variables. The `var` block declares a collection of uninitialized local variables. Function calls are an extra kind of expression:

Exp → ... | *Id* `(` *Exp*`,`...`,`*Exp* `)`

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

Functions as Values

We also allow functions as first-class values. The name of a function can be used as a kind of variable that refers to the function, and such function values can be assigned to ordinary variables, passed as arguments to functions, and returned from functions.

We add a generalized form of function calls (sometimes called *computed* or *indirect* function calls, in contrast to the simple *direct* calls described earlier):

Exp → ... | *Exp* `(` *Exp* `,` ... `,` *Exp* `)`

Unlike simple function calls, the function being called is now an expression that evaluates to a function value. Function values allow us to illustrate the main challenges that arise with methods in object-oriented languages and with higher-order functions in functional languages.

The following example program contains three functions:

```
twice(f, x) {
   return f(f(x));
}

inc(y) {
  return y+1;
}

main(z) {
  return twice(inc, z);
}
```

In the `main` function, the `inc` function is passed as argument to the `twice` function, which calls the given function twice.

Pointers

To be able to build data structures and dynamically allocate memory, we introduce pointers:

Exp → ...
| `alloc` *Exp*
| `&` *Id*
| `*` *Exp*
| `null`

The first expression allocates a new cell in the heap initialized with the value of the given expression and results in a pointer to the cell. The second expression creates a pointer to a program variable, and the third expression dereferences a pointer value (this is also called a load operation). In order to assign values through pointers we allow another form of assignment (called a store operation):

Stm → ... | `*` *Exp* `=` *Exp* `;`

In such an assignment, if the expression on the left-hand-side evaluates to a pointer to a cell, then the value of the right-hand-side expression is stored in that cell. Pointers and integers are distinct values, and pointer arithmetic is not possible.

The following example illustrates the various pointer operations:

```
x = alloc null;
y = &x;
*x = 42;
z = **y;
```

The first line allocates a cell initially holding the value null, after the second line `y` points to the variable `x`, the third line assigns the value 42 to the cell allocated in the first line (thereby overwriting the null value), and the fourth line reads the new value of that cell via two pointer dereferences.

Records

A record is a collection of fields, each having a name and a value. The syntax for creating records and for reading field values looks as follows:

Exp → …
| `{` *Id* `:` *Exp* `,` … `,` *Id* `:` *Exp* `}`
| *Exp* `.` *Id*

Here is an example:

```
x = {f: 1, g: 2};
y = x.f;
```

The first line creates a record with two fields: one with name `f` and value 1, and one with name `g` and value 2. The second line reads the value of the `f` field.

To update the values of record fields we allow two other forms of assignment, for writing directly to a record held by a variable or indirectly via a pointer, respectively:

Stm → …
| *Id* `.` *Id* `=` *Exp* `;`
| `(` `*` *Exp* `)` `.` *Id* `=` *Exp* `;`

Consider this example:

```
x = {f: 1, g: 2};
y = &x;
x.f = 3;
(*y).g = 4;
```

Here, `x` holds a record, `y` holds a pointer to the same record, and the last two assignments update the values of the fields `f` and `g` of the record.

Records are passed by value, so, for example, if `x` holds a record then a simple assignment `z = x;` copies the record to `z`. For simplicity, the values of record fields cannot themselves be records (but they can be, for example, pointers to records).

Programs

A complete program is just a collection of functions:

Prog → *Fun* … *Fun*

(We sometimes also refer to individual 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.

We often use meta-variables $I \in Int$, $X \in Id$, $E \in Exp$, $S \in Stm$, $F \in Fun$, $P \in Prog$, sometimes with subscripts, ranging over the different syntactic categories.

To keep the presentation short, we deliberately have not specified all details of the TIP language, either of the syntax or 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.

In Chapter 12 we formally define semantics for a part of TIP.

2.2 Example Programs

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

```
ite(n) {
  var f;
  f = 1;
  while (n>0) {
    f = f*n;
    n = n-1;
  }
  return f;
}
```

The second program is recursive:

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

The third program is unnecessarily complicated:

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

```
  }
  return f;
}

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

More example programs can be found in the Scala implementation of TIP.

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 `*`*Id* rather than the more general `*`*Exp*, and similarly, function calls are always of the form *Id*`(`*Id*`,...,`*Id*`)` rather than *Exp*`(`*Exp*`,...,`*Exp*`)`. It may also be useful to flatten arithmetic expressions, arguments to direct calls, branch conditions, and return expressions.

As an example,

```
x = f(y+3)*5;
```

can be normalized to

```
t1 = y+3;
t2 = f(t1);
x = t2*5;
```

where `t1` and `t2` are fresh variables, so that each statement performs only one operation.

Exercise 2.2: Argue that any TIP program can be normalized so that all expressions, with the exception of right-hand side expressions of assignments, are variables. (This is sometimes called *A-normal form* [FSDF93].)

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

Exercise 2.4: Assume we want to normalize pointer operations such that load operations are restricted to statements of the form *Id* = * *Id* ; and store operations are restricted to statements of the form **Id* = *Id* ;. Explain how the statement `**x=**y;` can be normalized to this restricted syntax.

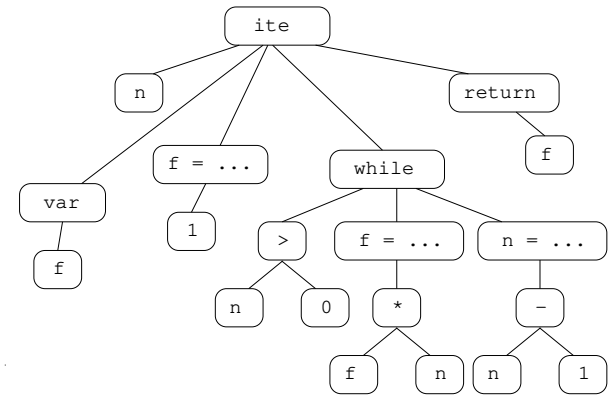
TIP uses lexical scoping; however, we make the notationally simplifying assumption that all declared variable and function names are unique in a program, i.e. that no identifier is declared more than once.

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

For real programming languages, we often use variations of the intermediate representations of compilers or virtual machines as foundation for implementing analyzers, rather than using the high-level source code.

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* analyses, for example, type analysis (Chapter 3), control flow analysis (Chapter 10), and pointer analysis (Chapter 11). 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 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 code. This idea goes back to the very first program analyzers in optimizing compilers [All70].

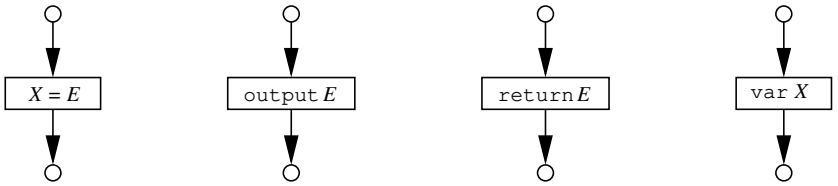
We first consider the subset of the TIP language consisting of a single function body without pointers. Control flow graphs for programs comprising multiple functions are treated in Chapters 8 and 10.

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.

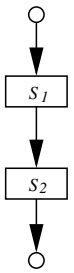
If v is a node in a CFG then $pred(v)$ denotes the set of predecessor nodes and $succ(v)$ the set of successor nodes.

For programs that are fully normalized (cf. Section 2.3), each node corresponds to only one operation.

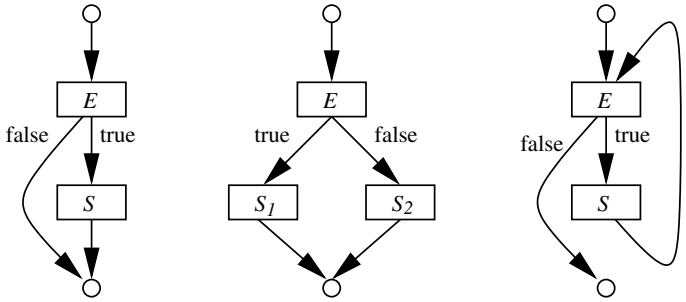
For now, we only consider simple statements, for which CFGs may be constructed in an inductive manner. The CFGs for assignments, `output`, `return` statements, and declarations look as follows:



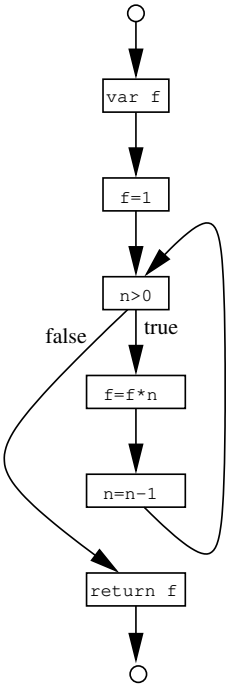
For the sequence $S_1\ S_2$, we eliminate the exit node of S_1 and the entry node of S_2 and glue the statements together:



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 kinds of values. 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 the correct number of arguments;
- the unary * operator only applies to pointers (or `null`);
- field accesses are only performed on records, not on other types of values; and
- the fields being accessed are guaranteed to be present in the records.

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 a nontrivial question, we immediately know (Section 1.3) 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 analysis will be conservative and reject some programs that in fact will not violate any requirements during execution.

In many mainstream statically typed languages, programmers provide type annotations for declarations. Type annotations serve as useful documentation, and they also make it easier to design and implement type systems. TIP does not have type annotations, so our type analysis must infer all the types based on how the variables and functions are being used in the program.

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 nontrivial, every instruction in your program should be reachable by some input.

Exercise 3.2: Even popular programming languages may have static type systems that do not by themselves guarantee type safety. Inform yourself about Java's covariant typing of arrays. Construct an example Java program that passes all of `javac`'s static type checks but would perform a type-unsafe array store if no runtime check were made. What happens when the program is executed, and why is the runtime check necessary?

The type analysis presented in this chapter is a simplified variant of Hindley–Milner type inference [Hin69, Mil78], which forms the basis of the type systems of many programming languages, including ML and OCaml, and has strongly influenced languages such as Haskell. Our constraint-based approach is inspired by Wand [Wan87].

To simplify the presentation, we postpone treatment of records until Section 3.4, and we discuss other possible extensions in Section 3.5.

The initial values of local variables are undefined in TIP programs; however, for this type analysis we assume that all the variables are initialized before they are used. In other words, this analysis is sound only for programs that never read from uninitialized variables. (A separate analysis that is designed specifically to check whether that property is satisfied is presented in Section 5.9.)

3.1 Types

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

$$
\begin{array}{rl}
\mathit{Type} \to & \mathsf{int} \\
\mid & \uparrow \mathit{Type} \\
\mid & (\ \mathit{Type}\ , \ldots, \mathit{Type}\ ) \to \mathit{Type}
\end{array}
$$

These type terms describe respectively integers, pointers, and functions. As an example, we can assign the type $(\mathsf{int})\to\mathsf{int}$ to the `ite` function from Section 2.2 and the type $\uparrow\mathsf{int}$ to the first parameter `p` of the `foo` function. Each kind of term is characterized by a *term constructor* with some arity. For example, $\uparrow$ is a term constructor with arity 1 as it has one sub-term, and the arity of a function type constructor $(\ldots)\to\ldots$ 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. (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:

$(\uparrow\mathsf{int},(\uparrow\mathsf{int},(\uparrow\mathsf{int},(\uparrow\mathsf{int},\ldots)\rightarrow\mathsf{int})\rightarrow\mathsf{int})\rightarrow\mathsf{int})\rightarrow\mathsf{int}$

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

Type $\rightarrow \ldots$
| μ *TypeVar* . *Type*
| *TypeVar*

TypeVar $\rightarrow \mathsf{t} \mid \mathsf{u} \mid \ldots$

We use meta-variables $\tau \in$ *Type* and $\alpha \in$ *TypeVar*, often with subscripts, ranging over types and type variables. A type of the form $\mu\alpha.\tau$ 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:

$\mu\mathsf{t}.(\uparrow\mathsf{int},\mathsf{t})\rightarrow\mathsf{int}$

Exercise 3.3: Explain how regular types can be represented by finite automata so that two types are equal if their automata accept the same language. Show an automaton that represents the type $\mu\mathsf{t}.(\uparrow\mathsf{int},\mathsf{t})\rightarrow\mathsf{int}$.

We allow free type variables (i.e., type variables that are not 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(a,b) {
  *b = a;
  return 0;
}
```

It has type $(\mathsf{t},\uparrow\mathsf{t})\rightarrow\mathsf{int}$ where t is a free type variable meaning that it can be any type, which corresponds to the polymorphic behavior of the function. Note that such type variables are not necessarily entirely unconstrained: the type of `a` may be anything, but it must match the type of whatever `b` points to. The more restricted type $(\mathsf{int},\uparrow\mathsf{int})\rightarrow\mathsf{int}$ is also a valid type for the `store` function, but we are usually interested in the most general solutions.

Note that the present analysis is whole-program and generates one global system of constraints. A free type variable in the final most-general solution is therefore not instantiated independently at each use; rather, it may be replaced

[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 $\mu\alpha$. 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.

by any type consistently throughout the solution. We return to this distinction when discussing let-polymorphism in Section 3.5.

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 $((\mathsf{int})\to\mathsf{int})\to(\mathsf{int},\mathsf{int})\to\mathsf{int}$.

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 local variable, function parameter, and 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 expression, statement, and function in the given program:

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

The notation '*op*' here represents any of the binary operators, except == which has its own rule. In the rule for `null`, α denotes a fresh type variable. (The purpose of this analysis is not to detect potential `null` pointer errors, so this simple model of `null` suffices.) Note that program variables and `var` blocks do not yield any constraints and that parenthesized expressions are not present in the abstract syntax.

For the program

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

we obtain the following constraints:

$[\![\texttt{short}]\!] = ()\rightarrow[\![\texttt{z}]\!]$
$[\![\texttt{input}]\!] = \mathsf{int}$
$[\![\texttt{x}]\!] = [\![\texttt{input}]\!]$
$[\![\texttt{alloc x}]\!] = \uparrow[\![\texttt{x}]\!]$
$[\![\texttt{y}]\!] = [\![\texttt{alloc x}]\!]$
$[\![\texttt{y}]\!] = \uparrow[\![\texttt{x}]\!]$
$[\![\texttt{z}]\!] = [\![\texttt{*y}]\!]$
$[\![\texttt{y}]\!] = \uparrow[\![\texttt{*y}]\!]$

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 pointers.

For a complete program, we add constraints to ensure that the types of the parameters and the return value of the `main` function are int:

$$\texttt{main(}X_1\texttt{,...,}X_n\texttt{)\{ ...return }E\texttt{; \}}: \quad [\![X_1]\!] = \ldots [\![X_n]\!] = [\![E]\!] = \mathsf{int}$$

In this way, a given program (or fragment of a 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. The order by which the AST is traversed is irrelevant.

Terms with different constructors (or function constructors of different arities) are unequal. All term constructors furthermore satisfy the general term equality axiom:

$$c(\tau_1, \ldots, \tau_n) = c(\tau'_1, \ldots, \tau'_n) \implies \tau_i = \tau'_i \text{ for each } i$$

where c is a term constructor and each τ_i and τ'_i is a sub-term. In the previous example two of the constraints are $[\![\texttt{y}]\!] = \uparrow [\![\texttt{x}]\!]$ and $[\![\texttt{y}]\!] = \uparrow [\![\texttt{*y}]\!]$, so by the term equality axiom we also have $[\![\texttt{x}]\!] = [\![\texttt{*y}]\!]$.

Furthermore, as one would expect for an equality relation, we have reflexitivity, symmetry, and transitivity:

$$\tau_1 = \tau_1$$

$$\tau_1 = \tau_2 \implies \tau_2 = \tau_1$$

$$\tau_1 = \tau_2 \ \wedge \ \tau_2 = \tau_3 \implies \tau_1 = \tau_3$$

for all terms τ_1, τ_2, and τ_3.

A *solution* assigns a type to each type variable, such that all equality constraints are satisfied.[2] 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 identifiers in the `short` program is the following:

$[\![\texttt{short}]\!] = ()\rightarrow\mathsf{int}$
$[\![\texttt{x}]\!] = \mathsf{int}$
$[\![\texttt{y}]\!] = \uparrow\mathsf{int}$
$[\![\texttt{z}]\!] = \mathsf{int}$

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

[2]We can define precisely what we mean by "solution" as follows. A *substitution* is a mapping σ from type variables to types. Applying a substitution σ to a type τ, denoted $\tau\sigma$, means replacing each free type variable α in τ by $\sigma(\alpha)$. A *solution* to a set of type constraints is a substitution σ where $\tau_1\sigma$ is identical to $\tau_2\sigma$ for each of the type constraints $\tau_1 = \tau_2$.

Exercise 3.8: Give a reasonable definition of what it means for one solution to be "more general" than another. (See page 21 for an example of two types where one is more general than the other.)

Exercise 3.9: This exercise demonstrates the importance of the term equality axiom. First explain what the following TIP code does when it is executed:

```
var x,y;
x = alloc 1;
y = alloc (alloc 2);
x = y;
```

Then generate the type constraints for the code, and apply the unification algorithm (by hand).

Exercise 3.10: 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 we use is based on the familiar union-find data structure (also called a disjoint-set data structure) from 1964 for representing and manipulating equivalence relations [GF64]. 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:[3]

- 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.

[3]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.

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, 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 unifying 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(τ2^r, τ1^r)
        else if τ1^r and τ2^r are proper types with same type constructor then
            Union(τ1^r, τ2^r)
            for each pair of sub-terms τ1' and τ2' of τ1^r and τ2^r, respectively do
                Unify(τ1', τ2')
            end for
        else
```

```
            unification failure
        end if
    end if
end procedure
```

Unification fails if attempting to unify two terms with different constructors (where function type 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 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 (i.e., terms that are not type variables) take precedence over type variables for becoming canonical representatives, so that it always suffices to consider only the canonical representative instead of all terms in the equivalence class.

Reading the solution after all constraints have been processed is then easy. For each program variable or expression that has an associated type variable, simply invoke FIND to find the canonical representative of its equivalence class. If the canonical representative has sub-terms (for example, in the term $\uparrow\tau$ we say that τ is a sub-term), find the solution recursively for each sub-term. The only complication arises if this recursion through the sub-terms leads to an infinite type, in which case we introduce a μ term accordingly.

Exercise 3.11: 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 (cf. Exercise 3.8).

(The most general solution, when one exists, for a program expression is also called the *principal type* of the expression.)

The unification solver only needs to process each constraint once. This means that although we conceptually first generate the constraints and then solve them, in an implementation we might as well interleave the two phases and solve the constraints on-the-fly, as they are being generated.

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

$[\![\texttt{foo}]\!] = ([\![\texttt{p}]\!],[\![\texttt{x}]\!])\to[\![\texttt{f}]\!]$
$[\![\texttt{*p}]\!] = \mathsf{int}$
$[\![\texttt{1}]\!] = \mathsf{int}$
$[\![\texttt{p}]\!] = \uparrow[\![\texttt{*p}]\!]$
$[\![\texttt{alloc 0}]\!] = \uparrow[\![\texttt{0}]\!]$
$[\![\texttt{q}]\!] = \uparrow[\![\texttt{*q}]\!]$
$[\![\texttt{f}]\!] = [\![\texttt{(*p)*(x(q,x))}]\!]$
$[\![\texttt{x(q,x)}]\!] = \mathsf{int}$
$[\![\texttt{input}]\!] = \mathsf{int}$
$[\![\texttt{n}]\!] = [\![\texttt{input}]\!]$
$[\![\texttt{foo}]\!] = ([\![\texttt{\&n}]\!],[\![\texttt{foo}]\!])\to[\![\texttt{foo(\&n,foo)}]\!]$
$[\![\texttt{(*p)-1}]\!] = \mathsf{int}$
$[\![\texttt{*p==0}]\!] = \mathsf{int}$
$[\![\texttt{f}]\!] = [\![\texttt{1}]\!]$
$[\![\texttt{0}]\!] = \mathsf{int}$
$[\![\texttt{q}]\!] = [\![\texttt{alloc 0}]\!]$
$[\![\texttt{q}]\!] = \uparrow[\![\texttt{(*p)-1}]\!]$
$[\![\texttt{*p}]\!] = \mathsf{int}$
$[\![\texttt{(*p)*(x(q,x))}]\!] = \mathsf{int}$
$[\![\texttt{x}]\!] = ([\![\texttt{q}]\!],[\![\texttt{x}]\!])\to[\![\texttt{x(q,x)}]\!]$
$[\![\texttt{main}]\!] = ()\to[\![\texttt{foo(\&n,foo)}]\!]$
$[\![\texttt{\&n}]\!] = \uparrow[\![\texttt{n}]\!]$
$[\![\texttt{*p}]\!] = [\![\texttt{0}]\!]$
$[\![\texttt{foo(\&n,foo)}]\!] = \mathsf{int}$

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

$[\![\texttt{p}]\!] = \uparrow\mathsf{int}$
$[\![\texttt{q}]\!] = \uparrow\mathsf{int}$
$[\![\texttt{alloc 0}]\!] = \uparrow\mathsf{int}$
$[\![\texttt{x}]\!] = \mu\mathrm{t}.(\uparrow\mathsf{int},\mathrm{t})\to\mathsf{int}$
$[\![\texttt{foo}]\!] = \mu\mathrm{t}.(\uparrow\mathsf{int},\mathrm{t})\to\mathsf{int}$
$[\![\texttt{\&n}]\!] = \uparrow\mathsf{int}$
$[\![\texttt{main}]\!] = ()\to\mathsf{int}$

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

Exercise 3.13: Consider this fragment of the example program shown earlier:

```
x = input;
*y = x;
z = *y;
```

Explain step-by-step how the unification algorithm finds the solution, including how the union-find data structure looks in each step.

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

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

creates these constraints:

$$[\![\texttt{null}]\!] = \uparrow \mathrm{t}$$
$$[\![\texttt{alloc null}]\!] = \uparrow [\![\texttt{null}]\!]$$
$$[\![\mathrm{p}]\!] = [\![\texttt{alloc null}]\!]$$
$$[\![\mathrm{p}]\!] = \uparrow [\![\mathrm{p}]\!]$$

which for $[\![\mathrm{p}]\!]$ have the solution $[\![\mathrm{p}]\!] = \mu\mathrm{t}.\uparrow\mathrm{t}$ that can be unfolded to $[\![\mathrm{p}]\!] = \uparrow\uparrow\uparrow\ldots$.

Exercise 3.14: Show what the union-find data structure looks like for the above example program.

Exercise 3.15: Generate and solve the constraints for the `ite` example program from Section 2.2.

Exercise 3.16: 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 null;
    *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 Record Types

To extend the type analysis to also work for programs that use records, we first extend the language of types with *record types*:

$$\mathit{Type} \rightarrow \ldots \mid \{\ \mathit{Id} : \mathit{Type}\ , \ldots, \mathit{Id} : \mathit{Type}\ \}$$

For example, the record type {a:int,b:int} describes records that have two fields, a and b, both with integer values. Record types with different sets of field names are considered as different term constructors.

Our goal is to be able to check conservatively whether the different kinds of errors listed in the beginning of the chapter may appear in the program being analyzed. This means that we need to distinguish records from other kinds of values. Specifically, we want the analysis to check that field accesses are only

performed on records, not on other types of values, and that the fields being accessed are present.

As a first attempt, the type constraints for record construction and field lookup can be expressed as follows, inspired by our treatment of pointers and dereferences. (Field write operations are handled in Exercise 3.19.)

$$\begin{array}{rl} \{\, X_1 : E_1, \ldots, X_n : E_n \,\}: & [\![\{\, X_1 : E_1, \ldots, X_n : E_n \,\}]\!] = \{X_1 : [\![E_1]\!], \ldots, X_n : [\![E_n]\!]\} \\ E.X: & [\![E]\!] = \{\ldots, X : [\![E.X]\!], \ldots\} \end{array}$$

Intuitively, the constraint rule for field lookup says that the type of E must be a record type that contains a field named X with the same type as $E.X$. The right-hand-side of this constraint rule is, however, not directly expressible in our language of types. One way to remedy this, without requiring any modifications of our unification algorithm, is to require that every record type contains all record fields that exist in the program, and add a special type term, $\diamond$, representing absent fields:

$$\mathit{Type} \to \ldots \mid \diamond$$

Let $F = \{f_1, f_2, \ldots, f_m\}$ be the set of all field names. We then use the following two constraint rules instead of the ones above.

$$\begin{array}{rl} \{\, X_1 : E_1, \ldots, X_n : E_n \,\}: & [\![\{\, X_1 : E_1, \ldots, X_n : E_n \,\}]\!] = \{f_1 : \gamma_1, \ldots, f_m : \gamma_m\} \\ & \text{where } \gamma_i = \begin{cases} [\![E_j]\!] & \text{if } f_i = X_j \text{ for some } j \in \{1, \ldots, n\} \\ \diamond & \text{otherwise} \end{cases} \\ & \text{for each } i = 1, 2, \ldots, m \\ \\ E.X: & [\![E]\!] = \{f_1 : \gamma_1, \ldots, f_m : \gamma_m\} \ \wedge \ [\![E.X]\!] \neq \diamond \\ & \text{where } \gamma_i = \begin{cases} [\![E.X]\!] & \text{if } f_i = X \\ \alpha_i & \text{otherwise} \end{cases} \\ & \text{for each } i = 1, 2, \ldots, m \end{array}$$

Each α_i denotes a fresh type variable. The constraints of the form $[\![E.X]\!] \neq \diamond$ can easily be checked separately after the unification process has been completed.

Exercise 3.17: Explain why it is easier to check the constraints of the form $[\![E.X]\!] \neq \diamond$ *after* instead of *during* the unification process (i.e., as soon as the other constraints related to field access operations are processed).

As an example, the two statements

```
x = {b: 42, c: 87};
y = x.a;
```

generate the following constraints, assuming that `a`, `b`, and `c` are the only fields in the program.

$[\![\texttt{x}]\!] = [\![\texttt{\{b: 42, c: 87\}}]\!]$
$[\![\texttt{\{b: 42, c: 87\}}]\!] = \{\text{a}:\diamond, \text{b}:[\![\texttt{42}]\!], \text{c}:[\![\texttt{87}]\!]\}$
$[\![\texttt{42}]\!] = \text{int}$
$[\![\texttt{87}]\!] = \text{int}$
$[\![\texttt{y}]\!] = [\![\texttt{x.a}]\!]$
$[\![\texttt{x}]\!] = \{\text{a}:[\![\texttt{x.a}]\!], \text{b}:x_1, \text{c}:x_2\}$
$[\![\texttt{x.a}]\!] \neq \diamond$

The type variables x_1 and x_2 in the solution for $[\![\texttt{x}]\!]$ reflect the fact that the fields `b` and `c` are irrelevant for the field lookup `x.a`. Similarly, $\diamond$ appears in the solution for $[\![\texttt{\{b: 42, c: 87\}}]\!]$, because the `a` field is absent in that record.

Exercise 3.18: Generate and solve the type constraints for the following program:

```
var a,b,c,d;
a = {f:3, g:17};
b = a.f;
c = {f:alloc 5, h:15};
d = c.f;
```

What happens if you change `c.f` to `c.g` in the last line?

Exercise 3.19: Specify suitable type constraint rules for both forms of field write statements, X_1`.`X_2`=`E`;` and `(*`E_1`).`X`=`E_2`;`. Then generate and solve the type constraints for the following program:

```
var a,b,c;
a = {f:null, g:17};
b = alloc {g:42, h:87};
a.f = b;
(*b).g = 117;
c = (*(a.f)).g;
```

Exercise 3.20: As mentioned in Section 2.1, the values of record fields in TIP programs cannot themselves be records. How can we extend the type analysis to check whether a given program satisfies this property?

Exercise 3.21: Assume we extend the TIP language with array operations. Array values are constructed using a new form of expressions (not to be confused with the syntax for records):

Exp → ... | `{` *Exp* `,` ... `,` *Exp* `}`

and individual elements are read and written as follows:

Exp → ... | *Exp* `[` *Exp* `]`
Stm → ... | *Exp* `[` *Exp* `]` `=` *Exp* `;`

For example, the following statements construct an array containing two integers, then overwrite the first one, and finally read both entries:

```
a = { 17, 87 };
a[0] = 42;
x = a[0] + a[1]; // x is now 129
```

Unlike records, arrays are constructed in the heap and passed by reference, so in the first line, the contents of the array are not copied, and `a` is like a pointer to the array containing the two integers.

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

Type → ... | *Type* `[]`

As an example, the type `int[][]` denotes arrays of arrays of integers.

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 program 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;
```

Exercise 3.22: As mentioned in Chapter 2, TIP does not have booleans as a separate type of values at runtime, but simply represents false as 0 and true as any other integer. Nevertheless, it can be useful to have a static type analysis that rejects expressions like `(x > y) * 17` and branches like `if (x * 42 - y)`, since programmers rarely intend to use results of comparisons in integer computations, or use results of integer computations as branch conditions. As a first step, let us extend our language of types with a new type, bool:

$$\mathit{Type} \to \ldots \mid \mathsf{bool}$$

How can we now change the type rules from page 22 such that the resulting type analysis rejects meaningless expressions like those above, but still accepts programs that programmers likely want to write in practice?
(See also Exercise 5.35, which is about a different approach to obtain such type information.)

Exercise 3.23: Discuss how TIP could be extended with strings and operations on strings, and how the type analysis could be extended accordingly to check, for example, that the string operations are only applied to strings and not to other types of values.

3.5 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 17;
  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 limitation, which is even more significant from a practical point of view, is the current treatment of polymorphic types. In fact, polymorphic types are not very useful in their current form. Consider this example program:

```
f(x) {
```

```
  return *x;
}

main() {
  return f(alloc 1) + *(f(alloc(alloc 2)));
}
```

It never causes an error at runtime but is not typable since, among other things, it generates constraints equivalent to

$$\uparrow\mathsf{int} = [\![\mathtt{x}]\!] = \uparrow\uparrow\mathsf{int}$$

which are clearly unsolvable. For this program, we could analyze the `f` function first, resulting in this polymorphic type:

$$[\![\mathtt{f}]\!] = (\uparrow\alpha)\rightarrow\alpha$$

When analyzing the `main` function, at each call to `f` we could then instantiate the polymorphic type according to the type of the argument: At the first call, the argument has type $\uparrow\mathsf{int}$ so in this case we treat `f` as having the type $(\uparrow\mathsf{int})\rightarrow\mathsf{int}$, and at the second call, the argument has type $\uparrow\uparrow\mathsf{int}$ so here we treat `f` as having the type $(\uparrow\uparrow\mathsf{int})\rightarrow\uparrow\mathsf{int}$. This technique works smoothly for this particular program because it contains no recursive calls. This idea is called *let-polymorphism* (and this is essentially how Hindley-Milner-style type analysis actually works in ML and related languages). In Section 8.2 we shall see a related program-analysis idea, *context sensitivity*. The price of the increased precision of let-polymorphism in type analysis is that the worst-case complexity increases from almost-linear to exponential [KTU90, Mai90].

Even with let-polymorphism, our type analysis still rejects programs that require *polymorphic recursion*, where recursive calls use the function at different types. For example:

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

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

Type inference with polymorphic recursion is undecidable in the general case [Hen93, KTU93].

Exercise 3.24: Explain the runtime behavior of the `polyrec` program, and why it is unfairly rejected by our type analysis, and why let-polymorphism does not help.

Yet another limitation of the type system presented in this chapter is that it ignores many other kinds of 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 in this program is that `*p` denotes a stack cell that has "escaped" from the `baz` function. As we shall see in Section 11.7, such problems can instead be handled by other kinds of static analysis.

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. The connection between lattices and program analysis was established in the seminal work by Kildall, Kam and Ullman [Kil73, KU77].

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 ($\top$) 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 $\top$ 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.

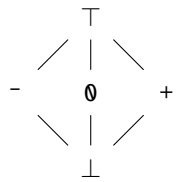
Consider this program:

```
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 $\top$ for that variable.

For this analysis we have an *abstract domain* consisting of the five abstract values $\{+, -, 0, \top, \bot\}$, 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 $\top$ represents the set of all integer values. Note that $\top$ 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, $\top$ 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 complete 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: $\forall x \in S \colon x \sqsubseteq x$
- transitivity: $\forall x, y, z \in S \colon x \sqsubseteq y \land y \sqsubseteq z \implies x \sqsubseteq z$
- anti-symmetry: $\forall x, y \in S \colon x \sqsubseteq y \land y \sqsubseteq x \implies x = y$

The abstract domain shown in Section 4.1 is an example of a partial order, where $S = \{\texttt{-}, \texttt{0}, \texttt{+}, \top, \bot\}$ and $\sqsubseteq$ specifies the ordering of the elements, for example, $\bot \sqsubseteq \texttt{+}$ and $\texttt{+} \sqsubseteq \top$.

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 partial order is a pair $(S, \sqsubseteq)$, but we often use the same name for the partial order and its underlying set S. Also, we sometimes write $y \sqsupseteq x$ ("y is greater than or equal x") instead of $x \sqsubseteq y$ ("x is less than or equal y").

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 $\bigsqcup X$, is defined by:

$$X \sqsubseteq \bigsqcup X \;\wedge\; \forall y \in S : X \sqsubseteq y \implies \bigsqcup X \sqsubseteq y$$

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

$$\bigsqcap X \sqsubseteq X \;\wedge\; \forall y \in S : y \sqsubseteq X \implies y \sqsubseteq \bigsqcap X$$

For pairs of elements, we sometimes use the infix notation $x \sqcup y$ (called the *join* of x and y) instead of $\bigsqcup\{x, y\}$ and $x \sqcap y$ (called the *meet* of x and y) instead of $\bigsqcap\{x, y\}$.

We also sometimes use the subscript notation, for example writing $\bigsqcup_{a \in A} f(a)$ instead of $\bigsqcup\{f(a) \mid a \in A\}$.

The least upper bound operation plays an important role in program analysis. As we shall see in Chapter 5, we use least upper bound when combining abstract information from multiple sources, for example when control flow merges after the branches of `if` statements.

Exercise 4.1: Let $X \subseteq S$. Prove that if $\bigsqcup X$ exists, then it must be unique. (A similar argument shows that the same property holds for greatest lower bounds: if $\bigsqcap X$ exists, then it must be unique.)

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

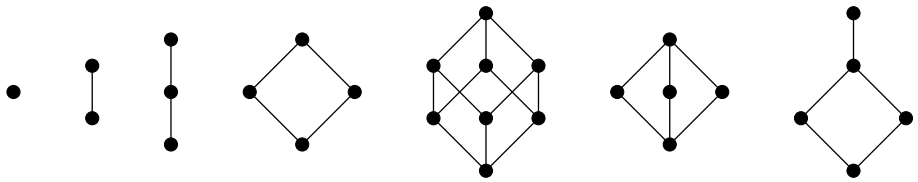
A *lattice* is a partial order $(S, \sqsubseteq)$ in which $x \sqcup y$ and $x \sqcap y$ exist for all $x, y \in S$. A *complete lattice* is a partial order $(S, \sqsubseteq)$ in which $\bigsqcup X$ and $\bigsqcap X$ exist for all $X \subseteq S$. Trivially, every complete lattice is a lattice. Most lattices we encounter in program analysis are complete lattices.

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

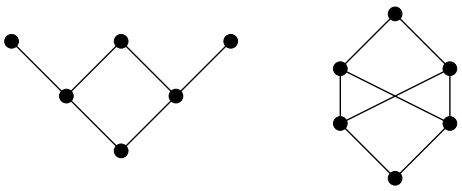
Exercise 4.4: Prove that if $(S, \sqsubseteq)$ is a nonempty finite lattice (i.e., a lattice where S is nonempty and finite), then $(S, \sqsubseteq)$ is also a complete lattice.

Exercise 4.5: Give an example of a nonempty lattice that is not a complete 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:



Exercise 4.6: Why do these two diagrams not define lattices?

It follows from the following exercise that to show that a partial order is a complete lattice, it suffices to argue that a least upper bound exists for each subset of its elements; the presence of greatest lower bounds then comes for free (and the dual property also holds).

Exercise 4.7: Prove that if S is a partially ordered set, then every subset of S has a least upper bound if and only if every subset of S has a greatest lower bound. (Hint: $\bigsqcap X = \bigsqcup\{y \in S \mid y \sqsubseteq X\}$.)

Every complete lattice has a unique *largest* element denoted $\top$ (pronounced *top*) and a unique *smallest* element denoted $\bot$ (pronounced *bottom*).

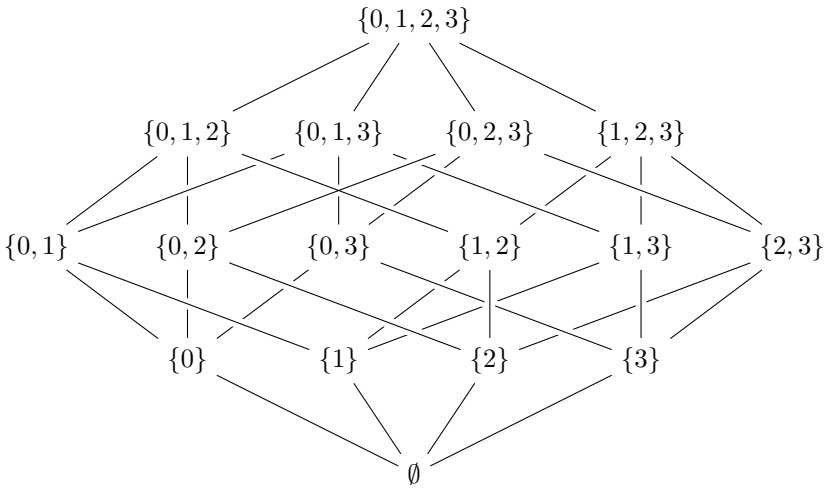
Exercise 4.8: Assume S is a partial order where $\bigsqcup S$ and $\bigsqcap S$ exist. Prove that $\bigsqcup S$ and $\bigsqcap S$ are the unique largest element and the unique smallest element, respectively, in S. In other words, we have $\top = \bigsqcup S$ and $\bot = \bigsqcap S$.

Exercise 4.9: Assume S is a partial order where $\bigsqcup S$, $\bigsqcap S$, $\bigsqcup \emptyset$ and $\bigsqcap \emptyset$ exist. Prove that $\bigsqcup S = \bigsqcap \emptyset$ and that $\bigsqcap S = \bigsqcup \emptyset$. (Together with Exercise 4.8 we then have $\top = \bigsqcap \emptyset$ and $\bot = \bigsqcup \emptyset$.)

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 (see Section 6.1).

4.3 Constructing Lattices

Every set $A = \{a_1, a_2, \dots\}$ defines a complete lattice $(\mathcal{P}(A), \subseteq)$, 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, $\{0, 1, 2, 3\}$, the powerset lattice looks like this:

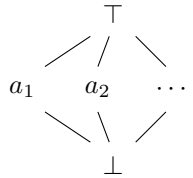


The above powerset lattice has height 4. In general, the lattice $(\mathcal{P}(A), \subseteq)$ has height $|A|$. We use powerset lattices in Chapter 5 to represent sets of variables or expressions.

The *reverse powerset lattice* for a set A is the complete lattice $(\mathcal{P}(A), \supseteq)$.

Exercise 4.10: Draw the Hasse diagram of the reverse powerset lattice for the set $\{\texttt{foo}, \texttt{bar}, \texttt{baz}\}$.

If $A = \{a_1, a_2, \dots\}$ is a set, then $\mathit{flat}(A)$ illustrated by



is a complete lattice with height 2. As an example, the set $\mathit{Sign} = \{+, -, \texttt{0}, \top, \bot\}$ with the ordering described in Section 4.1 forms a complete lattice that can also be expressed as $\mathit{flat}(\{+, \texttt{0}, -\})$.

If $L_1, L_2, \ldots, L_n$ are complete 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 order $\sqsubseteq$ is defined componentwise:[1]

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

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

Exercise 4.11: Show that the $\sqcup$ and $\sqcap$ operators for a product lattice $L_1 \times L_2 \times \ldots \times L_n$ can be computed componentwise (i.e. in terms of the $\sqcup$ and $\sqcap$ operators from L_1, L_2, …, L_n).

Exercise 4.12: Show that $\mathit{height}(L_1 \times \ldots \times L_n) = \mathit{height}(L_1) + \ldots + \mathit{height}(L_n)$.

If A is a set and L is a complete lattice, then we obtain a complete lattice called a *map* lattice consisting of the set of functions from A to L, ordered pointwise:[2]

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

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

We have already seen that the set $\mathit{Sign} = \{\texttt{+}, \texttt{-}, \texttt{0}, \top, \bot\}$ with the ordering described in Section 4.1 forms a complete lattice that we use for describing abstract values in the sign analysis. An example of a map lattice is $\mathit{StateSign} = \mathit{Var} \to \mathit{Sign}$ where Var 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 $\mathit{ProgramSign} = \mathit{StateSign}^n$ 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. 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.

Exercise 4.13: 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).

Exercise 4.14: Show that if A is finite and L has finite height then the height of the map lattice $A \to L$ is $\mathit{height}(A \to L) = |A| \cdot \mathit{height}(L)$.

[1] 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 $\top$, $\bot$.

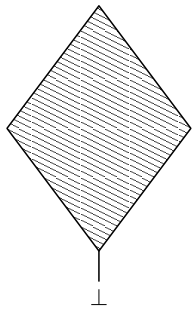
[2] The notation $[a_1 \mapsto x_1, a_2 \mapsto x_2, \ldots]$ means the function that maps a_1 to x_1, a_2 to x_2, etc.

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

Exercise 4.15: Argue that every product lattice L^n is isomorphic to the map lattice $A \to L$ whenever $|A| = n$.

Note that by Exercise 4.15 the lattice $\mathit{StateSign}^n$ is isomorphic to $\mathit{Node} \to \mathit{StateSign}$ where Node is the set of CFG nodes, so which of the two variants we use when describing the sign analysis is only a matter of preference.

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



It has $\mathit{height}(\mathit{lift}(L)) = \mathit{height}(L) + 1$ if L has finite height. One use of lifted lattices is for defining the lattice used in interval analysis (Section 6.1), another is for representing reachability information (Section 8.2).

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 equality constraints (equations) with one constraint variable for each program variable and line number from the program:[3]

$a_1 = \top$
$b_1 = \top$
$a_2 = +$

[3]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.

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

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 $\mathit{StateSign}$ from Section 4.3:[4]

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

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 substitution. However, mutually recursive equations may appear, for example for programs that contain loops (see Section 5.1).

Also notice that it is important for the analysis of this simple program that the order of statements is taken into account, which is called *flow-sensitive* analysis. Specifically, when a is read in line 3, the value comes from the assignment to a in line 2, not from the one in line 4.

Exercise 4.16: Give a solution to the constraint system above (that is, values for $x_1, \ldots, x_4$ that satisfy the four equations).

Exercise 4.17: 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 $f\colon L_1 \to L_2$ where L_1 and L_2 are lattices is *monotone* (or *order-preserving*) when $\forall x, y \in L_1 : x \sqsubseteq y \implies f(x) \sqsubseteq f(y)$. 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".

[4]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 4.18: A function $f\colon L \to L$ where L is a lattice is *extensive* when $\forall x \in L\colon x \sqsubseteq f(x)$. Assume L is the powerset lattice $\mathcal{P}(\{0,1,2,3,4\})$. 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 4.19: Prove that every constant function is monotone.

Exercise 4.20: A function $f\colon L_1 \to L_2$ where L_1 and L_2 are lattices is *distributive* when $\forall x, y \in L_1\colon f(x) \sqcup f(y) = f(x \sqcup y)$.

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

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

Exercise 4.21: Prove that a function $f\colon L_1 \to L_2$ where L_1 and L_2 are lattices is monotone if and only if $\forall x, y \in L_1\colon f(x) \sqcup f(y) \sqsubseteq f(x \sqcup y)$.

Exercise 4.22: Prove that function composition preserves monotonicity. That is, if $f\colon L_1 \to L_2$ and $g\colon L_2 \to L_3$ 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 $f\colon L_1 \times L_2 \to L_3$ where L_1, L_2, and L_3 are lattices is monotone when $\forall x_1, y_1 \in L_1, x_2 \in L_2\colon x_1 \sqsubseteq y_1 \implies f(x_1, x_2) \sqsubseteq f(y_1, x_2)$ and $\forall x_1 \in L_1, x_2, y_2 \in L_2\colon x_2 \sqsubseteq y_2 \implies f(x_1, x_2) \sqsubseteq f(x_1, y_2)$.

Exercise 4.23: The operators $\sqcup$ and $\sqcap$ can be viewed as functions. For example, $x_1 \sqcup x_2$ takes as input $x_1, x_2 \in L$ and returns an element from L as output. Show that $\sqcup$ and $\sqcap$ are monotone.

Exercise 4.24: Let $f\colon L^n \to L^n$ be a function with n arguments over a lattice L. We can view such a function in different ways: either as a function with n arguments from L, or as a function with a single argument from the product lattice L^n. Argue that this does not matter for the definition of monotonicity.

Exercise 4.25: 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 4.26: Recall that $f[a \mapsto x]$ denotes the function that is identical to f except that it maps a to x. Assume $f\colon L_1 \to (A \to L_2)$ and $g\colon L_1 \to L_2$ 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 $h\colon L_1 \to (A \to L_2)$ 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 $f\colon L \to L$ 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 complete lattice. An *equation system*[5] 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 $x_1, \ldots, x_n$ are variables and $f_1, \ldots, f_n\colon L^n \to L$ are functions which we call *constraint 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, $f\colon L^n \to L^n$,

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

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 4.27: 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

$$\begin{aligned} x_1 &= [\mathsf{a} \mapsto \top, \mathsf{b} \mapsto \top] \\ x_2 &= x_1[\mathsf{a} \mapsto +] \end{aligned}$$

[5] We also use the broader concept of *constraint systems*. An equation system is a constraint system where all constraints are equalities. On page 49 we discuss other forms of constraints.

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

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

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

Exercise 4.28: Show that the four constraint functions $f_1, \ldots, f_4$ are monotone. (Hint: see Exercise 4.26.)

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 4.29: Argue that your solution from Exercise 4.16 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 following *fixed-point theorem* due to Kleene [Kle52]:[6]

> In a complete lattice L with finite height, every monotone function $f\colon L \to L$ has a unique least fixed point denoted $\mathit{lfp}(f)$ defined as:
>
> $$\mathit{lfp}(f) = \bigsqcup_{i \geq 0} f^i(\bot)$$

(Note that when applying this theorem to the specific equation system shown above, f is a function over the product lattice L^n.)

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 $\mathit{lfp}(f) = f^k(\bot)$. Assume now that x is another fixed point. Since $\bot \sqsubseteq x$ it follows that $f(\bot) \sqsubseteq f(x) = x$, since f is

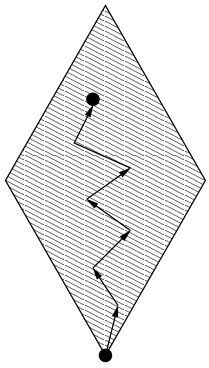
[6] The ordinary Kleene fixed-point theorem requires f to be continuous, meaning that $f(\bigsqcup A) = \bigsqcup_{a \in A} f(a)$ for every $A \subseteq L$ where A is totally ordered, i.e., $\forall x, y \in A : x \sqsubseteq y \vee y \sqsubseteq x$; on the other hand it does not require L to be a complete lattice but it can be any complete partial order (possibly with infinite height); see also Exercise 12.8.

monotone, and by induction we get that $\mathit{lfp}(f) = f^k(\bot) \sqsubseteq x$. Hence, $\mathit{lfp}(f)$ is a least fixed point, and by anti-symmetry of $\sqsubseteq$ it is also unique.

The theorem is a powerful result: It tells us not only that equation systems over complete 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 and study the connection to semantics in more detail in Chapter 12.

Exercise 4.30: Explain step-by-step how the naive fixed-point algorithm computes the solution to the equation system from Exercise 4.16.

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 4.31: Does the fixed-point theorem also hold without the assumption that f is monotone? If yes, give a proof; if no, give a counterexample.

Exercise 4.32: 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 counterexample. (Hint: see [CC79a].)

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 4.22 and 4.23). 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 $x = x \sqcap y$.

In case we have multiple inequations for each variable, those can also easily be reorganized, for example

$$
\begin{aligned}
x_1 &\sqsupseteq f_{1a}(x_1, \ldots, x_n) \\
x_1 &\sqsupseteq f_{1b}(x_1, \ldots, x_n)
\end{aligned}
$$

can be rewritten into

$$x_1 = x_1 \sqcup f_{1a}(x_1, \ldots, x_n) \sqcup f_{1b}(x_1, \ldots, x_n)$$

which again preserves the solutions.

Chapter 5

Dataflow Analysis with Monotone Frameworks

Classical dataflow analysis starts with a CFG and a complete 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 complete lattice and a space of monotone functions is called a *monotone framework* [KU77]. 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.

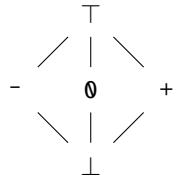
An analysis is *sound* if all solutions to the constraints correspond to correct information about the program. The solutions may be more or less imprecise, but computing the least solution will give the highest degree of precision possible. We return to the topic of analysis correctness and precision in Chapter 12.

Throughout this chapter we use the subset of TIP without function calls, pointers, and records; those language features are studied in Chapters 10 and 11 and in Exercise 5.10.

[1] As for type analysis, we will ambiguously use the notation $[\![S]\!]$ for $[\![v]\!]$ if S is the syntax associated with node 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

$$State = Var \to Sign$$

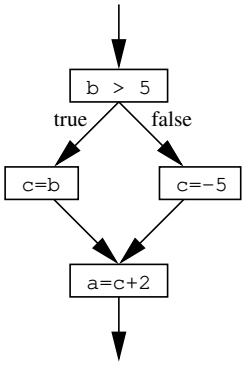
where Var is the set of variables occurring in the given program. Each element of this lattice can be thought of as an abstract state, hence its name. For each CFG node v we assign a constraint variable $[\![v]\!]$ denoting an abstract state that gives the sign values for all variables at the program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

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

Note that $JOIN(v)$ is a function of all the constraint variables $[\![v_1]\!], \ldots, [\![v_n]\!]$ for the program. 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:

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

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

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

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

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

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

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

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

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

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

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

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:

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

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.

A program with n CFG nodes, $v_1, \ldots, v_n$, is thus represented by n equations,

$$\begin{aligned}
[\![v_1]\!] &= af_{v_1}([\![v_1]\!], \ldots, [\![v_n]\!]) \\
[\![v_2]\!] &= af_{v_2}([\![v_1]\!], \ldots, [\![v_n]\!]) \\
&\vdots \\
[\![v_n]\!] &= af_{v_n}([\![v_1]\!], \ldots, [\![v_n]\!])
\end{aligned}$$

where $af_v : State^n \to State$ for each CFG node v.

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 4.22) from the $\sqcup$ operator (see Exercise 4.23), map updates (see Exercise 4.26), and the $eval$ function. The $sign$ function is constant (see Exercise 4.19). 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. (An informal argument suffices for now; we shall see a more formal approach to stating and proving this property in Section 12.4.)

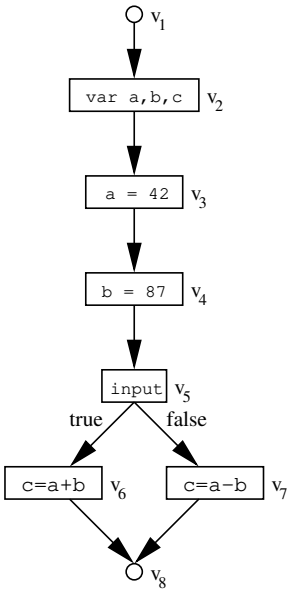
Exercise 5.5: The table for the abstract evaluation of == is unsound if we consider the full TIP language instead of the subset without pointers, function calls, and records. Why? And how could it be fixed?

Using the fixed-point algorithm from Section 4.4, we can now obtain the analysis result for the given program by computing $lfp(af)$ where $af(x_1, \ldots, x_n) = \big(af_{v_1}(x_1, \ldots, x_n), \ldots, af_{v_n}(x_1, \ldots, x_n)\big)$.

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 $\{v_1, \ldots, v_8\}$:

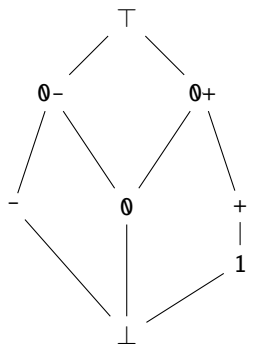


Exercise 5.6: Generate the equation system for this example program. Then solve the equations using the fixed-point algorithm from Section 4.4.

(Notice that the least upper bound operation is exactly what we need to model the merging of information at v_8!)

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 $\top$, 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, + divided by + results in $\top$ 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 (shown above) by means of 8×8 tables. Check that they are monotone. Does this new lattice improve precision for the expressions `(2>0)==1`, `x-x`, and `1/2`?

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;
```

Exercise 5.10: Explain how to extend the sign analysis to handle TIP programs that use records (see Chapter 2).

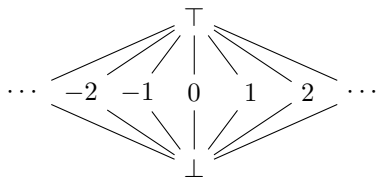
One approach, called *field insensitive* analysis, simply mixes together the different fields of each record. Another approach, *field sensitive* analysis, instead uses a more elaborate lattice that keeps different abstract values for the different field names.

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 $\top$. However, the resulting analysis will probably unfairly reject too many programs to be practical. Other more powerful analysis techniques, such as interval analysis (Section 6.1) and path sensitivity (Chapter 7) would be more useful for detecting such errors.

Notice that in this analysis we use the $\bigsqcup$ operation (in the definition of $JOIN$), but we never use the $\bigsqcap$ operation. In fact, when implementing analyses with monotone frameworks, it is common that $\bigsqcap$ is ignored entirely even though it mathematically exists.

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]



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

$$a \mathbin{\widehat{op}} b = \begin{cases} \bot & \text{if } a = \bot \text{ or } b = \bot \\ \bot & \text{if } op = / \text{ and } b = 0 \\ \top & \text{otherwise, if } a = \top \text{ or } b = \top \\ a \mathbin{op} b & \text{if } a \in \mathbb{Z} \text{ and } b \in \mathbb{Z} \end{cases}$$

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

Exercise 5.11: Argue that this definition of $\widehat{op}$ leads to a sound analysis. (An informal argument suffices; we shall see a more formal approach to proving soundness in Section 12.3.)

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;
```

This kind of optimization was among the first uses of static program analysis [Kil73].

Exercise 5.12: 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 $Node = \{v_1, v_2, \ldots, v_n\}$ we work in the complete 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 48), then gives us the desired solution for $[\![v_1]\!], \ldots, [\![v_n]\!]$.

Exercise 4.30 (page 49) 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 47) 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 5.13: 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 5.14: Continuing Exercise 4.30, 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 5.13). Second, we still apply all constraint functions in each iteration of the repeat-until loop. What matters for correctness, as should be clear from your solution to Exercise 5.13, is that the constraint functions are applied until the fixed point is reached for all of them. This observation leads to the *chaotic-*

iteration algorithm [Cou77, Kil73]:

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 $i \in \{1, \ldots, n\}$ such that $x_i \neq f_i(x_1, \ldots, x_n)$
    $x_i := f_i(x_1, \ldots, x_n)$
  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, naively computing the loop condition may now be more expensive than executing the loop body. However, *if* it terminates, the algorithm produces the right result.

Exercise 5.15: Prove that the chaotic-iteration algorithm computes the least fixed point of f, if it terminates. (Hint: see your solution to Exercise 5.13.)

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 4.28 and Exercise 5.6. We represent this information as a map

$$dep\colon Node \to \mathcal{P}(Node)$$

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, in particular, $dep(v_5) = \{v_6, v_7\}$. This means that whenever $[\![v_5]\!]$ changes its value during the fixed-point computation, only f_6 and f_7 may need to be recomputed.

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

procedure SimpleWorkListAlgorithm($f_1, \ldots, f_n$)
  $(x_1, \ldots, x_n) := (\bot, \ldots, \bot)$
  $W := \{v_1, \ldots, v_n\}$
  while $W \neq \emptyset$ **do**
    $v_i := W$.**removeNext**()
    $y := f_i(x_1, \ldots, x_n)$
    if $y \neq x_i$ **then**
      $x_i := y$
      for each $v_j \in dep(v_i)$ **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 5.16: 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 5.17: 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 5.18: 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 correct answers!)

Exercise 5.19: 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 5.20: 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 5.21: 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 5.22: 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 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

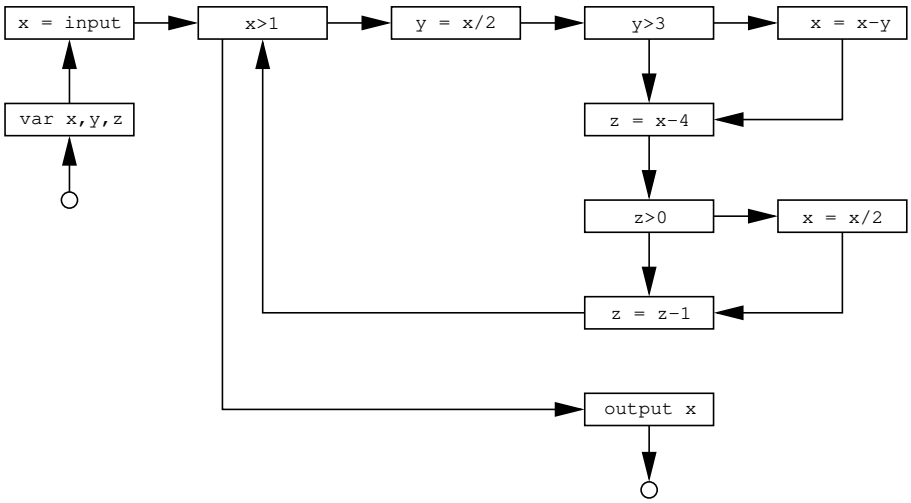
depends on the specific program being analyzed. For the example program

```
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:[4]

$$State = (\mathcal{P}(\{\texttt{x},\texttt{y},\texttt{z}\}), \subseteq)$$

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 will be conservative, since the computed set may be too large. We use the auxiliary definition

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

[4]A word of caution: For historical reasons, some textbooks and research papers, e.g. [Hec77, ALSU06], describe dataflow analyses using the lattices "upside down". This makes no difference whatsoever to the analysis results (because of the lattice dualities discussed in Chapter 4), but it can be confusing.

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

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

For all other nodes:

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

Exercise 5.24: Argue that the right-hand sides of the constraints define monotone functions.

The example program yields these constraints:

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

whose least solution is:

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

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 5.25: Consider the following program:

```
main() {
  var x,y,z;
  x = input;
  y = input;
  z = x;
  return 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 5.26: An analysis is distributive if all its constraint functions are distributive according to the definition from Exercise 4.20. Show that live variables analysis is distributive.

Exercise 5.27: As Exercise 5.25 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 5.25 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 using, 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 $\mathcal{P}(Var)^n$ has height $b \cdot n$, which bounds the number of iterations. Each lattice element consists of n sets, each of which can be represented as a bitvector of length b, requiring $b \cdot n$ bits in total. Using the observation from Exercise 5.19, we can ensure that $|succ(v)| \leq 2$ for every node v. Each iteration therefore performs $\mathcal{O}(n)$ union and difference operations on bitvectors of length b, as well as an equality test on the resulting lattice element, taking time $\mathcal{O}(b \cdot n)$ in total. Thus, the worst-case time complexity is $\mathcal{O}(b^2 \cdot n^2)$.

Exercise 5.28: 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 5.29: 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 5.30: Estimate the worst-case time complexity of the live variables analysis with the simple work-list algorithm, by using the formula from page 62. Can you obtain an asymptotically better bound by exploiting the specific structure of the live variables analysis?

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 no matter how the computation arrived at that program point, so we choose the lattice to be ordered by *reverse* subset inclusion.

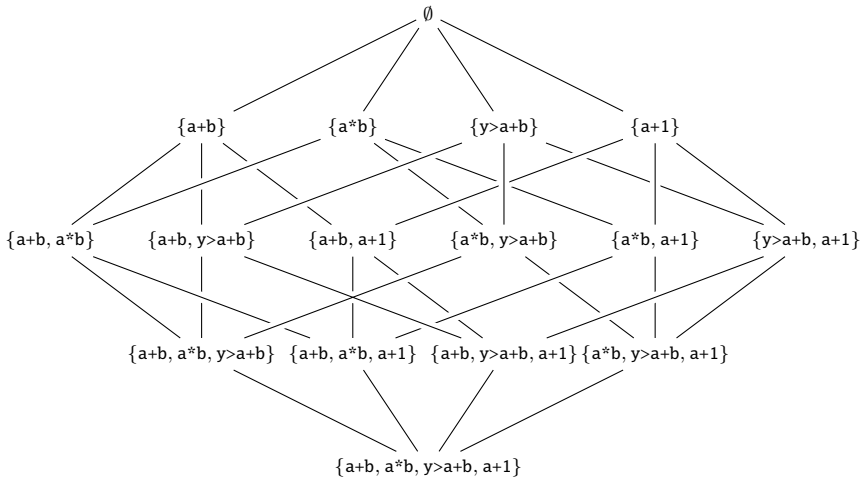
For the program

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

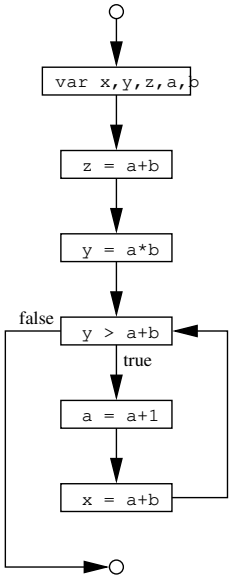
we have four different nontrivial expressions, so our lattice for abstract states is

$$State = (\mathcal{P}(\{\texttt{a+b}, \texttt{a*b}, \texttt{y>a+b}, \texttt{a+1}\}), \supseteq)$$

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 the above program looks as follows:



As usual in dataflow analysis, for each CFG node v we introduce a constraint variable $[\![v]\!]$ ranging over $State$. 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:

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

No expressions are available at entry nodes:

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

Branch conditions and output statements accumulate more available expressions:

`if (`E`):`
`while (`E`):`
`output` E`:` $\Big\}$ $\quad [\![v]\!] = \mathit{JOIN}(v) \cup \mathit{exps}(E)$

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

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

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

Exercise 5.31: Explain informally why the constraints are monotone and the analysis is sound.

For the example program, we generate the following constraints:

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

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

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

The expressions available at the program point *before* a node v can be computed from this solution as $\mathit{JOIN}(v)$. 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 5.32: 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, including 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:

$$State = (\mathcal{P}(\{\texttt{x=input}, \texttt{y=x/2}, \texttt{x=x-y}, \texttt{z=x-4}, \texttt{x=x/2}, \texttt{z=z-1}\}), \subseteq)$$

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

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

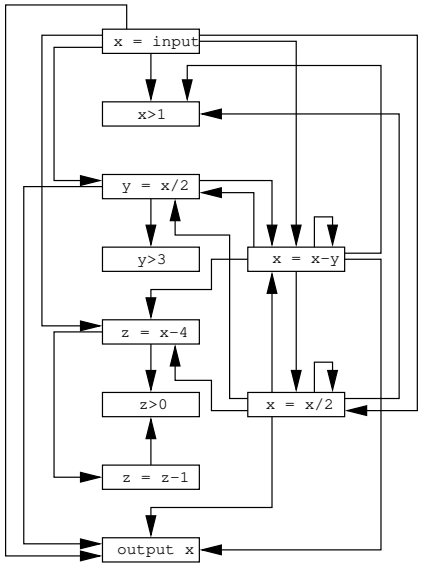
For assignments the constraint is:

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

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 5.33: 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 of the set of facts that hold. 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 of the set of facts that hold. 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 5.34: 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 5.35: Let us design a *flow-sensitive type analysis* for TIP. (This exercise can be seen as an alternative to the approach in Exercise 3.22.) 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. For the lattice for abstract states we choose

$$State = Var \to \mathcal{P}(\{\mathsf{integer}, \mathsf{boolean}\})$$

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 5.36: Assume we want to build an optimizing compiler for TIP (without pointers, function calls, and records). As part of this, we want to 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, `bool` 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 5.37: What is the $JOIN$ function for initialized variables analysis?

Exercise 5.38: Specify the constraint rule for assignments.

No statements other 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 54). However, for the entry node, we define $[\![entry]\!] = \emptyset$ since no variables are initialized initially (we ignore function parameters here).

Using the results from initialized variables analysis, a programming error detection tool could now check for every use of a variable, that the variable 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 5.39: 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 5.40: An alternative way to formulate initialized variables analysis would be to use the following map lattice instead of the powerset lattice:

$$State = Var \to Init$$

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

(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 analysis, it thus suffices to provide the transfer functions for all CFG nodes. As an example, in sign analysis where $L = Var \to Sign$, 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 8, 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 PropagationWorkListAlgorithm(t_1, ..., t_n)
    (x_1, ..., x_n) := (⊥, ..., ⊥)
    W := {v_1, ..., v_n}
    while W ≠ ∅ do
        v_i := W.removeNext()
        y := t_{v_i}(x_i)
        for each v_j ∈ dep(v_i) do
            z := x_j ⊔ y
            if x_j ≠ z then
                x_j := z
                W.add(v_j)
            end if
        end for
    end while
    return (x_1, ..., x_n)
end procedure
```

Compared to the simple work-list algorithm, this variant typically avoids many redundant least-upper-bound computations. 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 (hence the name of the algorithm) to all dependencies. Those that change are added to the work-list. We thereby have $t_v([\![v]\!]) \sqsubseteq [\![w]\!]$ for all nodes v, w where $w \in succ(v)$.

Exercise 5.41: Prove that PropagationWorkListAlgorithm computes the same solution as the other fixed-point solvers. (Hint: recall the discussion from page 49 about solving systems of inequations.)

Chapter 6

Widening

A central limitation of the monotone frameworks approach presented in Chapter 5 is the requirement that the lattices have finite height. In this chapter we describe a technique called *widening* that overcomes that limitation (and a related technique called *narrowing*), introduced by Cousot and Cousot [CC77].

6.1 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 6.2 to ensure convergence toward a fixed point.

The lattice describing a single abstract value is defined as

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

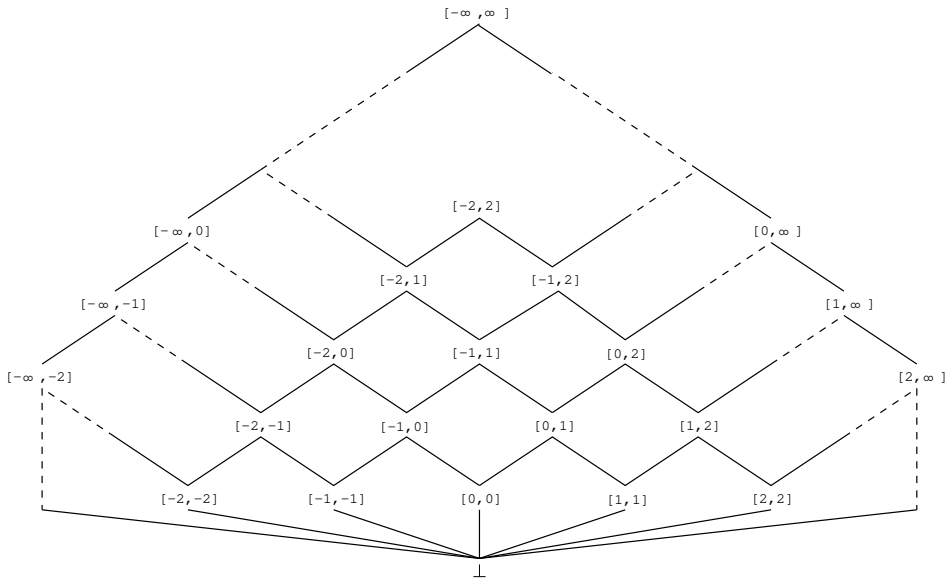
where

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

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

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

This lattice looks as follows:



This lattice does not have finite height, since it contains for example the following 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:

$$State = Var \to Interval$$

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

$eval(\sigma, X) = \sigma(X)$
$eval(\sigma, I) = [I, I]$
$eval(\sigma, \texttt{input}) = [-\infty, \infty]$
$eval(\sigma, E_1 \; op \; E_2) = \widehat{op}(eval(\sigma, E_1), eval(\sigma, E_2))$

The abstract operators are all defined by:

$$\widehat{op}([l_1, h_1], [l_2, h_2]) = \begin{cases} \bot & \text{if } D = \emptyset, \\ [\min D, \max D] & \text{otherwise,} \end{cases}$$

where $D = \{x \; op \; y \mid x \in [l_1, h_1],\ y \in [l_2, h_2],\ \text{and } x \; op \; y \text{ is defined}\}$. For example, $\widehat{+}([1, 10], [-5, 7]) = [1 - 5, 10 + 7] = [-4, 17]$.

Exercise 6.1: Explain informally why the definition of $eval$ given above is a conservative approximation compared to evaluating TIP expressions concretely. Give an example of how the definition of $eval$ could be modified to make the analysis more precise (and still sound).

Exercise 6.2: This general definition of $\widehat{op}$ looks simple in math, but it is non-trivial to implement it efficiently. 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!) It is acceptable to sacrifice optimal precision, but see how precise you can make it.

In Chapter 12 we provide a more formal treatment of the topics of soundness and precision.

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 6.3: Argue that the constraint functions are monotone.

In the preceding chapter, we defined the analysis result for each analysis as the least solution to the analysis constraints for the given program. Kleene's fixed-point theorem (Section 4.4, page 47) tells us that this is well-defined: a unique least solution always exists for such analyses. However, as the interval analysis defined in this section uses an infinite-height lattice, that fixed-point theorem does not apply, so the careful reader may wonder, do the interval analysis constraints actually have a least solution for any given program? The answer is affirmative.

One way to see that the least fixed point exists is to use a stronger variant of the fixed-point theorem that relies on transfinite iteration sequences and holds without the finite-height assumption [CC79a]; see also Exercise 4.32. Another approach is to use a different fixed-point theorem, known as Tarski's fixed-point theorem [Tar55]:[1]

> In a complete lattice L, every monotone function $f\colon L \to L$ has a unique least fixed point given by $lfp(f) = \bigsqcap\{x \in L \mid f(x) \sqsubseteq x\}$.

The proof is reasonably simple. Let $D = \{x \in L \mid f(x) \sqsubseteq x\}$ and $d = \bigsqcap D$. We first show that d is a fixed point of f, i.e., $f(d) = d$. Assume $x \in D$. Then

[1] Notice that Tarski's theorem does not require the lattice to have finite height; however, Kleene's fixed-point theorem (page 47) has the advantage (for finite-height lattices) that it is constructive in the sense that it directly leads to an algorithm for computing the least fixed point. The theorem presented here is a corollary of Tarski's more general theorem that the set of fixed points of f forms a complete lattice, but we do not need that more general property here.

$d \sqsubseteq x$ because d is a lower bound of D. By monotonicity of f we have $f(d) \sqsubseteq f(x)$, and $f(x) \sqsubseteq x$ because $x \in D$. Thus, $f(d) \sqsubseteq x$, so $f(d)$ is also a lower bound of D. Since d is the greatest lower bound of D we have $f(d) \sqsubseteq d$. By monotonicity of f we then have $f(f(d)) \sqsubseteq f(d)$. Therefore $f(d) \in D$, and since d is a lower bound of D, we have $d \sqsubseteq f(d)$. By anti-symmetry of $\sqsubseteq$ we get $f(d) = d$. To see that d is the unique least fixed point of f, assume d' is some fixed point of f, i.e., $f(d') = d'$. Then $d' \in D$, and since d is a lower bound of D, we get $d \sqsubseteq d'$, and, as usual, by anti-symmetry of $\sqsubseteq$ the least fixed point is unique.

Since the constraint functions for the interval analysis are monotone by Exercise 6.3, this theorem directly tells us that the least fixed point exists, thus the constraints for interval analysis always have a well-defined most precise solution. Nevertheless, we cannot in general compute the least fixed point using the fixed-point algorithms from Sections 4.4 and 5.3: For some programs, the fixed-point algorithms may never terminate, as the sequence of approximants

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

may not converge. A powerful technique to address this kind of problem is introduced in the next section.

Exercise 6.4: Give an example of a TIP program where the fixed-point algorithms from Sections 4.4 and 5.3 do not terminate for the interval analysis presented above.

6.2 Widening and Narrowing

To obtain convergence of the interval analysis presented in Section 6.1 we shall use a technique called *widening*. This technique generally works for any analysis that can be expressed using monotone equation systems, but it is typically used in flow-sensitive analyses with infinite-height lattices.

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

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

is guaranteed to converge on a fixed point that is 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 ω is monotone and extensive (see Exercise 4.18), and that the image $\omega(L) = \{\omega(x) \mid x \in L\}$ has finite height. The fixed-point algorithms can easily be adapted to use widening by applying ω in each iteration.

The widening function ω will intuitively coarsen the information sufficiently to ensure termination. For our interval analysis, ω is defined pointwise down to single intervals. It operates relative to a set B that consists of a finite set of integers together with $-\infty$ and ∞. Typically, B could be seeded with all the

integer constants occurring in the given program, but other heuristics could also be used. For single intervals we define the function $\omega' \colon \mathit{Interval} \to \mathit{Interval}$ by

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

$$\omega'(\bot) = \bot$$

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

As explained in Section 6.1, in the interval analysis the lattice L that the naive fixed-point algorithm works on is $L = \mathit{State}^n = (\mathit{Var} \to \mathit{Interval})^n$ where n is the number of nodes in the program CFG. The widening function $\omega \colon L \to L$ then simply applies ω' to every interval in the given abstract states:

$$\omega(\sigma_1, \ldots, \sigma_n) = (\sigma'_1, \ldots, \sigma'_n) \text{ where } \sigma'_i(X) = \omega'(\sigma_i(X)) \\ \text{for } i = 1, \ldots, n \text{ and } X \in \mathit{Var}$$

Exercise 6.5: Show that interval analysis with widening, with this definition of ω, always terminates and yields a solution that is a safe approximation of $\mathit{lfp}(f)$.

Hint: use Tarski's fixed-point theorem.

Widening is not only useful for infinite-height lattices; it can also be used as an acceleration technique for analyses that have finite-height lattices but converge too slowly and where a loss of precision is tolerable.

Widening generally shoots above the target, but a subsequent technique called *narrowing* may improve the result. Let f_ω denote the result of analysis with widening. A simple form of narrowing consists of computing $f(f_\omega)$. By Exercise 6.5 we have $\mathit{lfp}(f) \sqsubseteq f_\omega$. Then $f(f_\omega) \sqsubseteq \omega(f(f_\omega)) = (\omega \circ f)(f_\omega) = f_\omega$ since ω is extensive and f_ω is a fixed point of $\omega \circ f$. In other words, $f(f_\omega)$ is at least as precise as f_ω. Furthermore, $f(f(f_\omega)) \sqsubseteq f(f_\omega)$ since f is monotone, so $f(f_\omega) \in \{x \in L \mid f(x) \sqsubseteq x\}$ and thereby $\mathit{lfp}(f) \sqsubseteq f(f_\omega)$ by Tarski's fixed-point theorem. In other words, $f(f_\omega)$ is a safe approximation of $\mathit{lfp}(f)$. This technique may in fact be iterated any finite number of times, as seen in the following exercise.

Exercise 6.6: Show that $\forall i \colon \mathit{lfp}(f) \sqsubseteq f^{i+1}(f_\omega) \sqsubseteq f^i(f_\omega) \sqsubseteq f_\omega$.

An example will demonstrate the benefits of widening and narrowing. Consider this program:

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

Without widening, the naive fixed-point algorithm will produce the following diverging sequence of approximants for the program point after the `while`-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:

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

Exercise 6.7: Exactly how many narrowing steps are necessary to reach this solution?

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

$$f_\omega \sqsupseteq f(f_\omega) \sqsupseteq f^2(f_\omega) \sqsupseteq \dots$$

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

Exercise 6.8: Give an example of a TIP program where the narrowing sequence diverges for the interval analysis, when using widening followed by narrowing.

The simple kind of widening discussed above is sometimes unnecessarily aggressive: widening *every* interval in *every* abstract state in each iteration of the fixed-point algorithm is not necessary to ensure convergence. For this reason, traditional widening takes a more sophisticated approach that may lead to better analysis precision. It involves a binary operator, ∇:

$$\nabla \colon L \times L \to L$$

The widening operator ∇ (usually written with infix notation) must satisfy $\forall x, y \in L \colon x \sqsubseteq x \nabla y \wedge y \sqsubseteq x \nabla y$ (meaning that it is an upper bound operator) and that for any increasing sequence $z_0 \sqsubseteq z_1 \sqsubseteq z_2 \sqsubseteq \dots$, the sequence $y_0, y_1, y_2 \dots$

defined by $y_0 = z_0$ and $y_{i+1} = y_i \nabla z_{i+1}$ for $i = 0, 1, \ldots$ converges after a finite number of steps. With such an operator, we can approximate the least fixed point of f by computing the following sequence:

$$x_0 = \bot$$

$$x_{i+1} = x_i \nabla f(x_i)$$

This sequence eventually converges, that is, for some k we have $x_{k+1} = x_k$. Additionally, the result is a safe approximation of the ordinary fixed point: $\mathit{lfp}(f) \sqsubseteq x_k$.

Exercise 6.9: Prove that if ∇ is a widening operator (satisfying the criteria defined above), then $x_{k+1} = x_k$ and $\mathit{lfp}(f) \sqsubseteq x_k$ for some k.

This leads us to the following variant of the naive fixed-point algorithm with (traditional) widening:

procedure NAIVEFIXEDPOINTALGORITHMWITHWIDENING(f)
  $x := \bot$
  repeat
    $t := x$
    $x := x \nabla f(x)$
  until $x = t$
  return x
end procedure

The other fixed-point algorithms (Section 5.3) can be extended with this form of widening in a similar manner.

Note that if we choose as a special case $\nabla = \sqcup$, the computation of $x_0, x_1, \ldots$ proceeds exactly as with the ordinary naive fixed-point algorithm.

Exercise 6.10: Show that $\sqcup$ is a widening operator (albeit perhaps not a very useful one) if L has finite height.

The idea of using the binary widening operator ∇ is that it allows us to combine abstract information from the previous and the current iteration of the fixed-point computation (corresponding to the left-hand argument and the right-hand argument, respectively), and only coarsen abstract values that are unstable.

For the interval analysis we can for example define ∇ as follows. We first define a widening operator $\nabla' : \mathit{Interval} \times \mathit{Interval} \to \mathit{Interval}$ on single intervals:

$$\bot \nabla' y = y$$

$$x \nabla' \bot = x$$

$$[l_1, h_1] \nabla' [l_2, h_2] = [l_3, h_3]$$

where

$$l_3 = \begin{cases} l_1 & \text{if } l_1 \leq l_2 \\ max\{i \in B \mid i \leq l_2\} & \text{otherwise} \end{cases}$$

and

$$h_3 = \begin{cases} h_1 & \text{if } h_2 \leq h_1 \\ min\{i \in B \mid h_2 \leq i\} & \text{otherwise} \end{cases}$$

Compared to the definition of ω' for simple widening (see page 83), we now coarsen the interval end points only if they are unstable compared to the last iteration. Intuitively, an interval that does not become larger during an iteration of the fixed-point computation cannot be responsible for divergence.

Now we can define ∇ based on ∇', similarly to how we previously defined ω pointwise in terms of ω':

$$(\sigma_1, \ldots, \sigma_n) \nabla (\sigma'_1, \ldots, \sigma'_n) = (\sigma''_1, \ldots, \sigma''_n) \text{ where } \sigma''_i(X) = \sigma_i(X) \nabla' \sigma'_i(X) \text{ for } i = 1, \ldots, n \text{ and } X \in Var$$

Exercise 6.11: Show that this definition of ∇ for the interval analysis satisfies the requirements for being a widening operator.

With this more advanced form of widening but without using narrowing, for the small example program from page 83 we obtain the same analysis result as with the combination of simple widening and narrowing we looked at earlier.

Exercise 6.12: Explain why the "simple" form of widening (using the unary ω operator) is just a special case of the "traditional" widening mechanism (using the binary ∇ operator).

With the simple form of widening, the analysis effectively just uses a finite subset of L. In contrast, the traditional form of widening is fundamentally more powerful: Although each program being analyzed uses only finitely many elements of L, no finite-height subset suffices for all programs [CC92].

We can be even more clever by observing that divergence can only appear in the presence of recursive dataflow constraints (see Section 5.1) and apply widening only at, for example, CFG nodes that are loop heads.[2] In the above definition of ∇, this means changing the definition of σ''_i to

$$\sigma''_i(X) = \begin{cases} \sigma_i(X) \nabla' \sigma'_i(X) & \text{if node } i \text{ is a loop head} \\ \sigma'_i(X) & \text{otherwise} \end{cases}$$

[2] As long as we ignore function calls and only analyze individual functions, the loop heads are the `while` nodes in CFGs for TIP programs. If we also consider interprocedural analysis (Chapter 8) then recursive function calls must also be taken into account.

Exercise 6.13: Argue why applying widening only at CFG loop heads suffices for guaranteeing convergence of the fixed-point computation.

Then give an example of a program where this improves precision for the interval analysis, compared to widening at all CFG nodes.

Exercise 6.14: We can define another widening operator for interval analysis that does not require a set B of integer constants. In the definition of ∇' and ∇ from page 85, we simply change l_3 and h_3 as follows:

$$l_3 = \begin{cases} l_1 & \text{if } l_1 \leq l_2 \\ -\infty & \text{otherwise} \end{cases}$$

and

$$h_3 = \begin{cases} h_1 & \text{if } h_2 \leq h_1 \\ \infty & \text{otherwise} \end{cases}$$

Intuitively, this widening coarsens unstable intervals to $+/-\infty$.

(a) Argue that after this change, ∇ still satisfies the requirements for being a widening operator.

(b) Give an example of a program that is analyzed less precisely after this change.

Chapter 7

Path Sensitivity and Relational Analysis

Until now, we have ignored the values of branch and loop conditions by simply treating `if`- and `while`-statements as a nondeterministic choice between the two branches, which is called *control insensitive* analysis. Such analyses are also *path insensitive*, because they do not distinguish different paths that lead to a given program point. The information about branches and paths can be important for precision. 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.

Exercise 7.1: What would be the ideal (i.e., most precise, yet sound) analysis result for `x`, `y`, and `z` at the exit program point in the example above, when using the *Interval* lattice to describe abstract values? (Later in this chapter we shall see an improved interval analysis that obtains that result.)

7.1 Control Sensitivity using Assertions

To exploit the information available in conditionals, 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 statements 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:

$$[\![\texttt{assert(}E\texttt{)}]\!] = JOIN(v)$$

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 $E_1 \texttt{ > } E_2$ or $E_1 \texttt{ == } E_2$ 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 $X \texttt{ > } E$ and $E \texttt{ > } X$. The former kind of assertion can be handled by the constraint rule

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

where gt models the greater-than operator:

$$gt([l_1, h_1], [l_2, h_2]) = [l_1, h_1] \sqcap [l_2 + 1, \infty]$$

Exercise 7.2: Argue that this constraint for `assert` is sound and monotone.

Exercise 7.3: Specify a constraint rule for `assert(`$E \texttt{ > } X$`)`.

Negated conditions are handled in a similar fashion, 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 7.4: Discuss how more conditions may be given nontrivial constraints for `assert` to improve analysis precision further.

As the analysis now takes the information in the branch conditions into account, this kind of analysis is called *control sensitive* (or *branch sensitive*). An alternative approach to control sensitivity that does not involve `assert` statements is to model each branch node in the CFG using two constraint variables instead of just one, corresponding to the two different outcomes of the evaluation of the branch condition. Another approach is to associate dataflow constraints with CFG edges instead of nodes. The technical details of such approaches will be different from the approach taken here, but the overall idea is the same.

7.2 Paths and Relations

Control sensitivity is insufficient for reasoning about relational properties that can arise due to branches in the 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. (A more realistic setting with multiple files can be handled using techniques presented in Chapter 11.) 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, that `open` is only called if the file is currently closed, and that the file is definitely closed at the program exit. In this example program, the critical property is that the branch containing the call to `close` is taken only if the branch containing the call to `open` was taken earlier in the execution.

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

$$L = \mathcal{P}(\{\mathsf{open}, \mathsf{closed}\})$$

(The lattice order is implicitly $\subseteq$ for a powerset lattice.) For example, the lattice element $\{\mathsf{open}\}$ means that the file is definitely not closed, and $\{\mathsf{open}, \mathsf{closed}\}$ means that the status of the file is unknown. 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:

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

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

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

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

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

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 7.5: 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 of two powerset lattices that keeps track of both the status of the file and the value of the flag:

$$L' = \mathcal{P}(\{\mathsf{open}, \mathsf{closed}\}) \times \mathcal{P}(\{\texttt{flag} = 0, \texttt{flag} \neq 0\})$$

(The lattice order is implicitly defined as the componentwise subset ordering of the two powersets.) For example, the lattice element $\{\texttt{flag} \neq 0\}$ in the rightmost sub-lattice means that `flag` is definitely not 0, and $\{\texttt{flag} = 0, \texttt{flag} \neq 0\}$ means that the value of `flag` is unknown. 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 7.6: Specify the constraints that fit with the L' lattice. Then show that the analysis produces the lattice element $(\{\mathsf{open}, \mathsf{closed}\}, \{\texttt{flag} = 0, \texttt{flag} \neq 0\})$ at the program point 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. One approach to achieve this is 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

$$L'' = Path \to L$$

where $Path$ is a finite set of *path contexts*. A path context is typically a predicate over the program state.[1] (For instance, a condition expression in TIP defines such a predicate.) In general, each statement is then analyzed in $|Path|$ different path contexts, each describing a set of paths that lead to the statement, which is why this kind of analysis is called *path sensitive*. For the example above, we can use $Path = \{\texttt{flag} = 0, \texttt{flag} \neq 0\}$.

The constraints for `open`, `close`, and *entry* can now be defined as follows.[2]

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

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

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

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

$$\texttt{flag = 0}: \qquad [\![v]\!] = [\texttt{flag} = 0 \mapsto \textstyle\bigcup_{p \in Path} JOIN(v)(p), \quad \texttt{flag} \neq 0 \mapsto \emptyset]$$

[1] Another way to select $Path$ is to use sequences of branch nodes.

[2] We here use the lambda abstraction 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.

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

$$\texttt{flag = } E\text{:} \qquad [\![v]\!] = \lambda q. \textstyle\bigcup_{p \in Path} JOIN(v)(p)$$

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`:

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

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

Exercise 7.7: 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:

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

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:

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

$[\![\texttt{assert(flag)}]\!] = [\texttt{flag} \neq 0 \mapsto [\![\texttt{flag}]\!](\texttt{flag} \neq 0),\ \texttt{flag} = 0 \mapsto \emptyset]$
$[\![\texttt{close()}]\!] = \lambda p.\{\mathsf{closed}\}$
$[\![\texttt{assert(!flag)}]\!] = [\texttt{flag} = 0 \mapsto [\![\texttt{flag}]\!](\texttt{flag} = 0),\ \texttt{flag} \neq 0 \mapsto \emptyset]$
$[\![\mathit{exit}]\!] = \lambda p.\big([\![\texttt{close()}]\!](p) \cup [\![\texttt{assert(!flag)}]\!](p)\big)$

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

	$\texttt{flag} = 0$	$\texttt{flag} \neq 0$
$[\![\mathit{entry}]\!]$	{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}	$\emptyset$
$[\![\mathit{exit}]\!]$	{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 7.8: For the present example, the basic lattice L is defined as a powerset of a finite set $A = \{\mathsf{open}, \mathsf{closed}\}$.

(a) Show that $\mathit{Path} \to \mathcal{P}(A)$ is isomorphic to $\mathcal{P}(\mathit{Path} \times A)$ for any finite set A. (This explains why such analyses are called *relational*: each element of $\mathcal{P}(\mathit{Path} \times A)$ is a (binary) relation between Path and A.)

(b) Reformulate the analysis using $\mathcal{P}(\{\texttt{flag} = 0, \texttt{flag} \neq 0\} \times \{\mathsf{open}, \mathsf{closed}\})$ instead of L'' (without affecting the analysis precision).

Exercise 7.9: Describe a variant of the example program above where the present analysis would be improved by combining it with constant propagation.

In general, the program analysis designer is left with the choice of Path. Often, Path 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 $Path$ 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 [BR02].

Exercise 7.10: 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 7.11: 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 7.12: Construct yet another variant of the `open`/`close` example program where the desired property can only be established with a choice of $Path$ 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.)

Exercise 7.13: The following TIP code computes the absolute value of `x`:

```
if (x < 0) {
  sgn = -1;
} else {
  sgn = 1;
}
y = x * sgn;
```

Design an analysis (i.e., define a lattice and describe the relevant constraint rules) that is able to show that `y` is always positive or zero after the last assignment in this program.

Chapter 8

Interprocedural Analysis

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

8.1 Interprocedural Control Flow Graphs

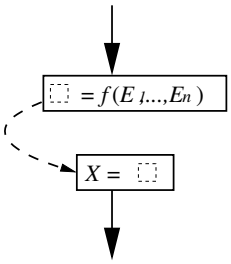
We use the subset of the TIP language containing functions, but still ignore pointers and functions as values. As we shall see, the CFG for an entire program is then quite simple to obtain. It becomes more complicated when adding function values, which we discuss in Chapter 10.

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:

$X = f(E_1, \ldots, E_n);$

Exercise 8.1: Show how any program can be normalized (cf. Section 2.3) to have this form.

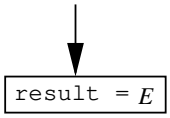
In the CFG, we represent each function call statement using *two* nodes: a *call node* representing the connection from the caller to the entry of `f`, and an *after-call node* where execution resumes after returning from the exit of `f`:



Next, we represent each `return` statement

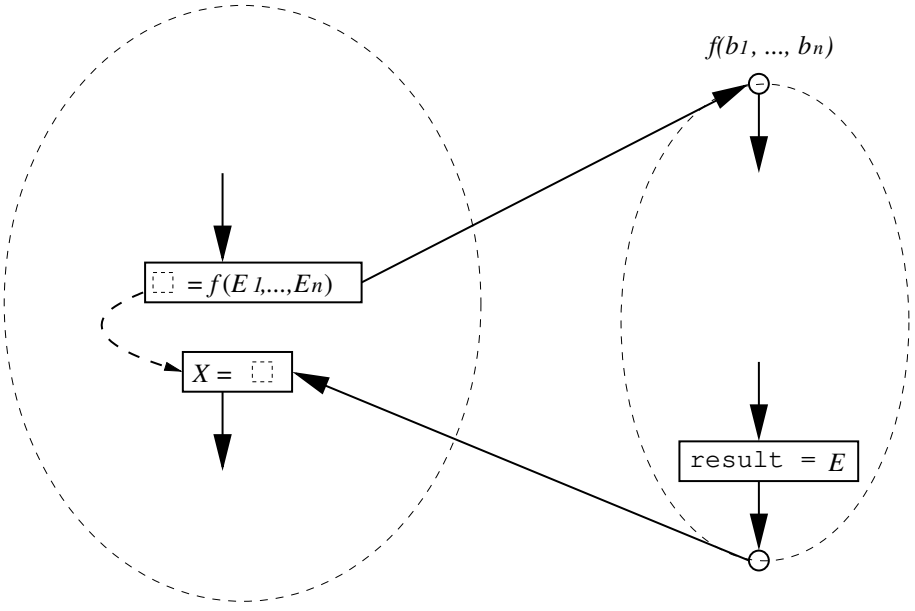
```
return E;
```

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:

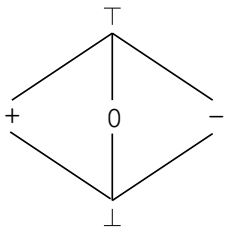


The connection between the call node and its after-call node is represented by a special edge (not in *succ* and *pred*), which we need for propagating abstract values for local variables of the caller.

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

Exercise 8.2: 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 $\mathit{State} = \mathit{Var} \to \mathit{Sign}$. 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 to model the information flow at function calls and returns. First, we treat the call nodes as no-ops that simply collect information from their predecessors. Thus, if v is a call node, we define $[\![v]\!] = \mathit{JOIN}(v)$. For an entry node v of a function $f(b_1, \ldots, b_n)$ 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 and $w \in \mathit{pred}(v)$ is a caller:

$$s_w \sqsubseteq [\![v]\!]$$

Intuitively, this shows how information flows from the call node (the left-hand-side of $\sqsubseteq$) to the function entry node (the right-hand-side of $\sqsubseteq$). This formulation fits well with algorithms like PROPAGATIONWORKLISTALGORITHM from Section 5.10.

Exercise 8.3: Explain why these two formulations of the constraint rule for function entry nodes are equivalent.

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

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:

$$[\![v]\!] = \bot[b_1 \mapsto \top, \ldots, b_n \mapsto \top]$$

Function exit nodes are modeled as no-ops, just like call nodes, so if v is an exit node we can define $[\![v]\!] = \mathit{JOIN}(v)$. For an after-call node v that stores the return value in the variable X and where v' is the accompanying call node and $w \in \mathit{pred}(v)$ 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.

Notice that in this definition of the analysis constraints 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 8.4: 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 8.5: 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, algorithms like PROPAGATIONWORKLISTALGORITHM (Section 5.10) are often preferred for interprocedural dataflow analysis.

Exercise 8.6: For the interprocedural sign analysis, how can we choose $\mathit{dep}(v)$ when v is a call node, an after-call node, a function entry node, or a function exit node?

8.2 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 $\top$. That value propagates through the body of `f` and back to the callers, so both `x` and `y` also become $\top$. 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 7.

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

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

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

in analyses where $State$ 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 $Context$ be a singleton set, which amounts to context-insensitive analysis. Another extreme we shall investigate is to pick $Context = State$, which allows *full* context sensitivity. These ideas originate from the work by Sharir and Pnueli [SP81].

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 $Context \to \mathit{lift}(State)$ rather than just $State$. For example, the constraint rule for assignments X=E in intraprocedural sign analysis from Section 5.1,

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

becomes

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

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.

8.3 Context Sensitivity with Call Strings

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

$$Context = Call^{\leq k}$$

where k is a positive integer. With this choice of call contexts, we can obtain a similar effect to function cloning or inlining, but without actually changing the CFG. The idea is that a tuple $(c_1, c_2, \ldots, c_m) \in Call^{\leq k}$ identifies the topmost m call sites on the call stack. If $(e_1, \ldots, e_n) \in (Context \to State)^n$ 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. The length of the tuples is bounded to ensure that $Context$ is

[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: $A^{\leq k} = \bigcup_{i=0,\ldots,k} A^i$.

finite. The context represented by the empty tuple, denoted ϵ, thus identifies the empty call stack when execution is initiated at the `main` function. Tuples $(c_1, c_2, \ldots, c_m)$ where $m < k$ identify call stacks of height exactly m (in which case c_m must be a call node in the `main` function), while tuples where $m = k$ identify call stacks of height at least m (intuitively, call strings longer than k are truncated).

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

Exercise 8.7: What is the height of the lattice $(\mathit{Context} \to \mathit{State})^n$ when $\mathit{Context} = \mathit{Call}^{\leq k}$ and $\mathit{State} = \mathit{Var} \to \mathit{Sign}$, expressed in terms of k (the call string bound), $n = |\mathit{Node}|$, and $b = |\mathit{Var}|$?

To demonstrate the call string approach we again consider sign analysis applied to the program from Section 8.2. 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 $\mathit{Context} = \{\epsilon, c_1, c_2\}$, so the analysis only tracks the top-most call site. We can now define the analysis constraints such that, in particular, at the entry of the function `f`, we obtain the lattice element

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

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.

Parameter passing at function calls is modeled in the same way as in context-insensitive analysis, but now taking the call contexts into account. Assume w is a call node and v is the entry node of the function f`(`b_1`,` $\ldots$ `,`b_n`)` being called. The abstract state $s_w^{c'}$ defined by

$$s_w^{c'} = \begin{cases} \mathsf{unreachable} & \text{if } [\![w]\!](c') = \mathsf{unreachable} \\ \bot[b_1 \mapsto \mathit{eval}([\![w]\!](c'), E_1^w), \ldots, b_n \mapsto \mathit{eval}([\![w]\!](c'), E_n^w)] & \text{otherwise} \end{cases}$$

describes the dataflow from w to v, like the context-insensitive variant but now parameterized with a context c' for the call node. This definition also models the fact that no new dataflow can appear from call node w in context c' if that combination of node and context is unreachable (perhaps because the analysis has not yet encountered any dataflow to that node and context).

The constraint rule for a function entry node v where $w \in \mathit{pred}(v)$ is a caller and $c' \in \mathit{Context}$ is a call context can then be written as follows.

$$s_w^{c'} \sqsubseteq [\![v]\!](c) \quad \text{where } c = w$$

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 c at the

function entry node v. In this simple case where $k = 1$, the call node w is directly used as context c for the function entry 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 from the call context).

Alternatively, the constraint rule can be expressed as an equation by collecting the abstract states for all relevant call nodes and contexts:

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

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.

Exercise 8.8: Verify that this constraint rule for function entry nodes indeed leads to the lattice element shown above for the example program.

Exercise 8.9: 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.)

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

$$[\![v]\!](c) = \begin{cases} \mathsf{unreachable} & \text{if } [\![v']\!](c) = \mathsf{unreachable} \ \vee \ [\![w]\!](v') = \mathsf{unreachable} \\ [\![v']\!](c)[X \mapsto [\![w]\!](v')(\texttt{result})] & \text{otherwise} \end{cases}$$

Notice that with this kind of context sensitivity, 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 8.10: Write and solve the constraints that are generated by the interprocedural sign analysis for the program from Exercise 8.4, 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 8.11: Assume we have analyzed a program P 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`:

$[c_2 \mapsto [\texttt{result} \mapsto \texttt{-}],$
$(c_1, c_2) \mapsto [\texttt{result} \mapsto \texttt{+}],$
all other contexts $\mapsto$ unreachable$]$

Explain informally what this tells us about the program P. Give an example of what program P may be.

Exercise 8.12: 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 $\top$ by the analysis if and only if $k < 2$.

Exercise 8.13: 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, it is common to select k individually for each call site, based on heuristics.

8.4 Context Sensitivity with the Functional Approach

Consider this variant of the program from Section 8.2:

```
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 distinguishing 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

$$Context = State$$

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

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

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

Exercise 8.14: What is the height of this lattice, expressed in terms of $h = height(State)$ and $s = |State|$?

The idea is that a lattice element for a CFG node v is a map $m_v \colon State \to \mathit{lift}(State)$ such that $m_v(s)$ approximates the possible states at v given that the current function containing v was entered in a state that matches s. The situation $m_v(s) = \mathsf{unreachable}$ 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_v 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 8.2 (page 103), we will now define the analysis constraints such that, in particular, we obtain the following lattice element at the exit of the function `f`:[3]

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

This information 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 context where `z` is - 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 $f\,(b_1, \ldots, b_n)$ and a call $w \in pred(v)$ to the function is the same as in the call strings approach, except for the condition on c:

$$s_w^{c'} \sqsubseteq [\![v]\!](c) \quad \text{where } c = s_w^{c'}$$

(The abstract state $s_w^{c'}$ is defined as in Section 8.3.) 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 $s_w^{c'}$. This makes sense because contexts are abstract states with the functional approach.

[3] We here use the map update notation described on page 44 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`.

Exercise 8.15: Verify that this constraint rule for function entry nodes indeed leads to the lattice element shown above for the example program.

Exercise 8.16: 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.)

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

$$[\![v]\!](c) = \begin{cases} \mathsf{unreachable} & \text{if } [\![v']\!](c) = \mathsf{unreachable} \ \vee \ [\![w]\!](s^c_{v'}) = \mathsf{unreachable} \\ [\![v']\!](c)[X \mapsto [\![w]\!](s^c_{v'})(\texttt{result})] & \text{otherwise} \end{cases}$$

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 8.17: 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 8.18: Write and solve the constraints that are generated by the interprocedural sign analysis for the program from Exercise 8.4, 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 8.19: 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 9

Distributive Analysis Frameworks

Recall from Exercise 5.26 that an analysis is distributive if all its constraint functions are distributive as defined in Exercise 4.20: A function $f\colon L_1 \to L_2$ where L_1 and L_2 are lattices is distributive when $\forall x, y \in L_1\colon f(x) \sqcup f(y) = f(x \sqcup y)$. In this chapter, we demonstrate how distributivity enables efficient algorithms for context- and flow-sensitive analysis. These results build on a combination of (1) the functional approach to context sensitivity (Section 8.4), and (2) a clever compact representation of distributive functions. The techniques presented in this chapter originate from Reps, Horwitz and Sagiv [RHS95, SRH96] but are presented in a constraint-based style to make the connections to the preceding chapters clearer.

9.1 Motivating Example: Possibly-Uninitialized Variables Analysis

The goal of possibly-uninitialized variables analysis is to approximate at each program point in a given program which variables may have values that come from uninitialized variables. Possibly-uninitialized variables analysis is very similar to *taint analysis*, which aims to infer which computations may involve "tainted" data that come from untrusted input.

As an example, the possibly-uninitialized variables at each program point in the following program are shown in comments:

```
var a,b,c;    // a,b,c
a = input;    // b,c
b = a + c;    // b,c
```

```
if (input) {  // b,c
  c = 1;      // b
}             // b,c
```

At the program point immediately after the variable declarations, all three variables are possibly-uninitialized. Notice that `b` remains possibly-uninitialized after the assignment `b = a + c` because its value depends on `c` (and therefore this analysis is not simply the dual of the initialized variables analysis from Section 5.9). At the final program point, `c` is possibly-uninitialized despite the assignment `c = 1` because the branch containing that assignment may not be taken.

We shall now design such an analysis that is both flow sensitive and context sensitive using the functional approach to context sensitivity, following the methodology from Chapters 5 and 8. As discussed in Section 8.4, the functional approach to context sensitivity is precise enough to completely avoid the problem with interprocedurally invalid paths. The results of such an analysis can be used as follows. When analyzing a program, we want to know for a given variable X and program point v inside a function f whether f may be called in some context c such that X is possibly-uninitialized at v in context c – and if so, a warning can be emitted informing the programmer that the program behavior is likely unintended if the instruction at v uses X.

First, we choose a suitable lattice of abstract states,

$$State = \mathcal{P}(Var)$$

with each abstract state denoting a set of possibly-uninitialized variables. With the functional approach to context sensitivity where contexts are themselves abstract states, we set $Context = State$ so that the analysis lattice for a program with n CFG nodes looks as follows:

$$\big(\mathcal{P}(Var) \to \mathit{lift}(\mathcal{P}(Var))\big)^n$$

An element of this lattice contains a function $m_v : \mathcal{P}(Var) \to (\mathcal{P}(Var) \cup \{\mathsf{unreachable}\})$, called a *jump function*, for each CFG node v. Such a function has the intuitive meaning that if the program function that contains v is entered from a call with possibly-uninitialized variables $s \in \mathcal{P}(Var)$, then the set of possibly-uninitialized variables at v is $m_v(s) \in \mathcal{P}(Var)$, and $m_v(s) = \mathsf{unreachable}$[1] means that no such call exists in the program (in other words, that f and thereby also v are unreachable in context s). For example, if the function `foo` below is called with `y` being possibly-uninitialized while `x` is definitely initialized, then `y` and `z` are possibly-uninitialized at the `return` instruction, thus $m_{\texttt{return z}}(\{\texttt{y}\}) = \{\texttt{y}, \texttt{z}\}$.

```
foo(x, y) {
  var z;
  z = x - y;
  z = z * z;
```

[1] unreachable here denotes the bottom element of the lifted lattice as in Section 8.2.

```
  return z;
}
```

The entry of the main function is always reachable with the empty context:[2]

$$[\![entry_{\texttt{main}}]\!](\emptyset) \neq \mathsf{unreachable}$$

The transfer function for a variable declaration, `var` X, models the fact that variables in TIP are uninitialized right after they have been declared:

$$t_{\texttt{var}\ X}(s) = s \cup \{X\}$$

The transfer function for an assignment, $X = E$, is defined as follows where $vars(E)$ denotes the set of variables occurring in E.

$$t_{X=E}(s) = \begin{cases} s \cup \{X\} & \text{if } vars(E) \cap s \neq \emptyset \\ s \backslash \{X\} & \text{otherwise} \end{cases}$$

As in Section 8.2, the analysis constraint for $[\![v]\!]$ when v is a variable declaration or an assignment can be expressed using the transfer function and the $JOIN$ function:

$$[\![v]\!](c) = \begin{cases} t_v(JOIN(v,c)) & \text{if } JOIN(v,c) \in \mathcal{P}(Var) \\ \mathsf{unreachable} & \text{if } JOIN(v,c) = \mathsf{unreachable} \end{cases}$$

where

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

Exercise 9.1: Explain intuitively why the above definition of the transfer function for assignments is correct for possibly-uninitialized variables analysis.

Exercise 9.2: Specify suitable constraint rules for other kinds of CFG nodes than variable declarations and assignments (most importantly, non-main function entry nodes and after-call nodes).

Finally, we can collect the possibly-uninitialized variables at each program point v by computing $\bigsqcup_{c \in Context} [\![v]\!](c)$.

[2] In TIP, the parameters of the main function obtain values from the program input, and other variables cannot be accessed outside the scope of their declaration.

Exercise 9.3: At each program point in the following program, which variables are possibly-uninitialized according to the context sensitive analysis described above? What is the result if instead using context insensitive analysis?

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

p(a,b) {
  if (a > 0) {
    b = input;
    a = a - b;
    b = p(a,b);
    output(a);
    output(b);
  }
  return b;
}
```

In Section 8.4 we saw that the jump function m_v for a function exit node v has the interesting property that it "summarizes" the entire function for all possible contexts the function may be called in, by mapping entry abstract states to exit abstract states. If, for example, v is the exit node of the `foo` function on page 112, then m_v is this function:

$$m_v(s) = \begin{cases} s \cup \{z\} & \text{if } x \in s \ \vee \ y \in s \\ s \backslash \{z\} & \text{otherwise} \end{cases}$$

This means that once $m_v(s)$ has been computed for some abstract state s, then the result can be reused for all calls to `foo` where the corresponding abstract state for the function entry is s, without revisiting the function body. The work-list algorithm from Section 5.3 automatically performs such reuse.

Another important observation from Section 8.4 is that the functional approach to context sensitivity can be computationally expensive when the number of contexts is large. The following exercise shows that a basic implementation of the analysis is not scalable, despite the reuse effect for calls with the same call context and the use of unreachable for avoiding analysis of functions in unreachable contexts.

Exercise 9.4: Estimate the worst-case complexity of possibly-uninitialized variables analysis as specified above. Assume n is the size of the program being analyzed (measured as the number of CFG nodes) and that we use either the naive fixed-point algorithm from Section 4.4 or the work-list algorithm from Section 5.3.

The key to achieve a more scalable solution is distributivity, as demonstrated in the following sections.

Exercise 9.5: Show that possibly-uninitialized variables analysis is distributive (as defined in Exercise 5.26).

9.2 An Alternative Formulation

So far, we have simply used the standard methodology from Chapters 5 and 8, without exploiting distributivity. But before we look into distributivity, let us first study an alternative formulation of possibly-uninitialized variables analysis, which is a step toward the IFDS framework described in Section 9.4. It is divided into two phases:

1. a *pre-analysis* that computes function summaries for all reachable functions, and
2. a *main analysis* that computes possibly-uninitialized variables for all CFG nodes, leveraging information from the pre-analysis at function calls.

Interestingly, both phases are context insensitive (not context sensitive!), and yet the resulting analysis achieves the same precision as the original formulation from Section 9.1 for computing which variables may be possibly-uninitialized at a given program point.

The pre-analysis phase We first describe the pre-analysis. It uses this lattice for a program with n CFG nodes:

$$\big(\mathit{lift}(\mathcal{P}(\mathit{Var}) \to \mathcal{P}(\mathit{Var}))\big)^n$$

Even though we think of this as a context insensitive analysis, the analysis lattice looks very similar to the one from page 112. In the old analysis, an element of the lattice for individual CFG nodes was a set of possibly-uninitialized program variables for each context. In this context insensitive pre-analysis, an element of the lattice for individual CFG nodes is instead either unreachable or a jump function $m_v \colon \mathcal{P}(\mathit{Var}) \to \mathcal{P}(\mathit{Var})$ with almost the same meaning as before, relating the set of possibly-uninitialized variables at the entry of the function containing v with the set of possibly-uninitialized variables at v. The lattice *lift* operation is used slightly differently; this new analysis does not track which

contexts each function may be called in, but only which functions may be called in some context, thereby providing a coarser notion of reachability.

With this choice of analysis lattice, the constraint rules for the different kinds of CFG nodes can be defined as follows.

The entry node $\mathit{entry}_{\texttt{main}}$ of the main function is always reachable, and for any given set s of possibly-uninitialized variables at the entry of the function, s is trivially the set of possibly-uninitialized variables at this node since it is the same node:

$$[\![\mathit{entry}_{\texttt{main}}]\!]_1(s) = s$$

The subscript at $[\![\cdot]\!]_1$ indicates that the constraint variables here are for analysis phase 1. For the entry node entry_f of any other function f, the constraint is the same, except that we account for reachability of the function:

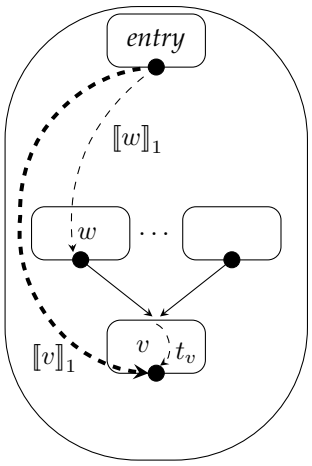
$$[\![\mathit{entry}_f]\!]_1(s) = s \qquad \text{if the program contains a call node } \square = f(\ldots) \text{ where } [\![\square = f(\ldots)]\!]_1 \neq \mathsf{unreachable}$$

If there are no reachable calls to f, we implicitly have $[\![\mathit{entry}_f]\!]_1 = \mathsf{unreachable}$ in the least solution to the constraints, because unreachable is the bottom element of the lattice. Notice that the only information that is carried from the caller to the callee in this pre-analysis is whether the function is reachable – there is no flow of information between the functions about which variables are possibly-uninitialized, unlike the first formulation of the possibly-uninitialized variables analysis.

The constraint for an assignment node $v\colon X = E$ is defined using function composition, where $t_{X=E}$ is the transfer function defined in Section 9.1:

$$[\![X = E]\!]_1 = t_{X=E} \circ \bigsqcup_{\substack{w \in \mathit{pred}(X=E) \wedge \\ [\![w]\!]_1 \neq \mathsf{unreachable}}} [\![w]\!]_1 \qquad \text{if } [\![w]\!]_1 \neq \mathsf{unreachable} \text{ for some } w \in \mathit{pred}(X=E)$$

Again, we implicitly have $[\![X = E]\!]_1 = \mathsf{unreachable}$ if the node has no reachable predecessors. For each reachable predecessor w in the CFG, $[\![w]\!]_1$ is a jump function for w. By computing the least upper bound of those functions and composing with the transfer function, we take one step further, thereby obtaining the jump function for v as illustrated with the thick dashed edge below.



Variable declaration nodes can be handled similarly, and as in Chapter 8, we simply treat `return` E instructions as assignments, `result` = E.

The constraints for after-call nodes are more involved, because this is where we model parameter passing and return values. For a call f`(`E_1`,` $\ldots$`,`E_n`)` to a function f`(`X_1`,` $\ldots$`,`X_n`) {` $\ldots$ `}` where $E_1, \ldots, E_n$ are the arguments and $X_1, \ldots, X_n$ are the parameters, let $t_{X_1,\ldots,X_n=E_1,\ldots,E_n}$ denote the transfer function for a generalized form of assignment that considers whether each expression $E_1, \ldots, E_n$ may involve possibly-uninitialized variables and binds the results to $X_1, \ldots, X_n$, respectively:

$$t_{X_1,\ldots,X_n=E_1,\ldots,E_n}(s) = (s \cup A_s) \backslash B_s$$

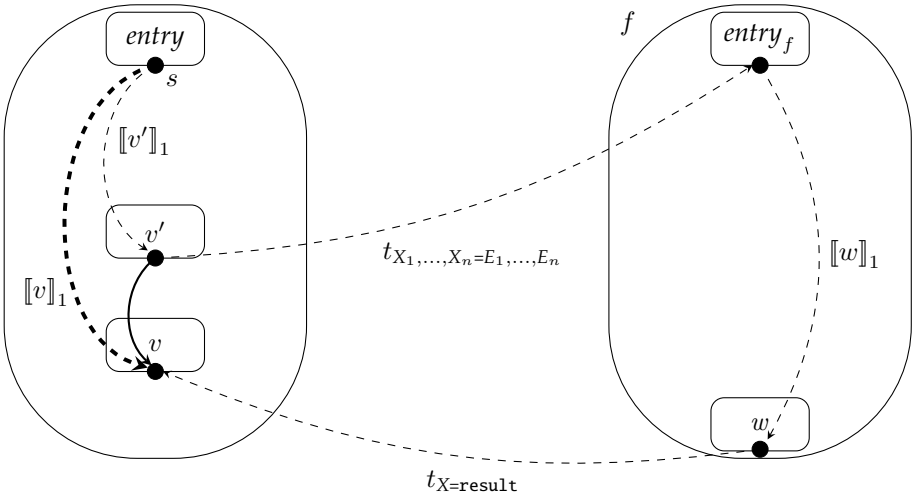
Here, $A_s = \{X_i \mid vars(E_i) \cap s \neq \emptyset\}$ is the set of parameters that are possibly-uninitialized and $B_s = \{X_i \mid vars(E_i) \cap s = \emptyset\}$ is the set of parameters that are definitely initialized.

Assume v is an after-call node $v\colon X$ `=` $\square$ where $v'\colon \square$ `=` f`(`E_1`,` $\ldots$`,`E_n`)` is its accompanying call node, the function f has parameters $X_1, \ldots, X_n$, and $w \in pred(v)$ is f's exit node. The jump function for v is the same as the one for v' except that the return value for X is transferred from `result` at w:

$$[\![X = \square]\!]_1(s) = [\![v']\!]_1(s) \backslash \{X\} \cup \left((t_{X=\texttt{result}} \circ [\![w]\!]_1 \circ t_{X_1,\ldots,X_n=E_1,\ldots,E_n} \circ [\![v']\!]_1)(s) \cap \{X\} \right)$$

Think of s as a set of variables that are possibly-uninitialized at the entry of the function containing v. The composition of the jump functions and transfer functions $t_{X=\texttt{result}} \circ [\![w]\!]_1 \circ t_{X_1,\ldots,X_n=E_1,\ldots,E_n} \circ [\![v']\!]_1$ defines a jump function for all interprocedurally valid execution paths[3] from the entry of the function containing v, through the function f via the call v', to v:

[3] As explained in Section 8.2, an interprocedurally valid path is a path in the CFG where all calls and returns match.



Exercise 9.6: What are suitable constraint rules for the remaining kinds of CFG nodes (for example, `if` E)?

The main analysis phase After the pre-analysis has been executed, jump functions are available for all CFG nodes. This makes it easy to compute the sets of possibly-uninitialized variables in the main analysis phase. This analysis uses the same lattice for abstract states as the first formulation of the analysis in Section 9.1:

$$State = \mathcal{P}(Var)$$

The analysis is flow sensitive but context insensitive, and we want to compute information only for functions that may be reachable, so we use this analysis lattice for a program with n CFG nodes:

$$\big(lift(\mathcal{P}(Var))\big)^n$$

By using a lifted lattice, we can distinguish between nodes that belong to unreachable functions and nodes that are in possibly reachable functions but with the empty set of possibly-uninitialized variables.

The entry of the main function is always reachable:

$$[\![entry_{\texttt{main}}]\!]_2 \neq \mathsf{unreachable}$$

The subscript at $[\![\cdot]\!]_2$ indicates that these constraint variables belong to analysis phase 2.

For the entry node $entry_f$ of a function f with parameters $X_1, \ldots, X_n$, the set of possibly-uninitialized variables is obtained from all the calls to f while taking reachability into account:

$$[\![entry_f]\!]_2 = \bigsqcup_{w \in pred(entry_f)} s_w$$

where

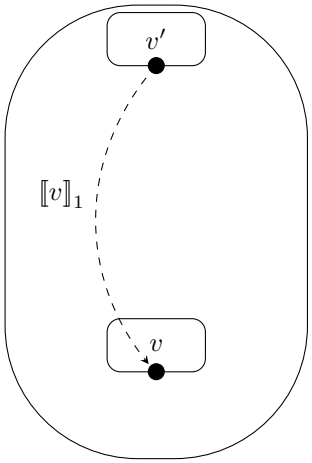
$$s_w = \begin{cases} \mathsf{unreachable} & \text{if } [\![w]\!]_2 = \mathsf{unreachable} \\ t^w_{X_1,\ldots,X_n=E^w_1,\ldots,E^w_n}([\![w]\!]_2) & \text{otherwise} \end{cases}$$

and $t^w_{X_1,\ldots,X_n=E^w_1,\ldots,E^w_n}$ models parameter passing for call node w: $f(E^w_1, \ldots, E^w_n)$ as before.

For all non-entry nodes we can now conveniently use the jump functions computed by the pre-analysis. If v is a non-entry node and v' is the entry of the function containing v, we have the following constraint that applies v's jump function $[\![v]\!]_1$ to the set of possibly-uninitialized variables at v' unless v' is unreachable:

$$[\![v]\!]_2 = \begin{cases} \mathsf{unreachable} & \text{if } [\![v']\!]_2 = \mathsf{unreachable} \\ [\![v]\!]_1([\![v']\!]_2) & \text{otherwise} \end{cases}$$

This can be illustrated as follows:



Exercise 9.7: Prove that the alternative two-phase formulation of possibly-uninitialized variables analysis has the same precision as the original formulation, in the sense that it computes the same sets of possibly-uninitialized variables for each program point.

Exercise 9.8: As a continuation of Exercise 9.4, prove that the alternative two-phase formulation of possibly-uninitialized variables analysis has the same worst-case complexity as the original formulation. (Hint: The bottleneck is the pre-analysis, not the main analysis.)

Despite the apparent lack of progress suggested by Exercises 9.7 and 9.8, this alternative formulation of the analysis paves the way for a trick presented in the next section.

9.3 Compact Representation of Distributive Functions

Certain distributive functions allow compact representation. In this section we consider two families of such functions:

1. functions of the form $f\colon \mathcal{P}(D) \to \mathcal{P}(D)$ that are distributive and where D is a finite set,
2. functions of the form $f\colon (D \to L) \to (D \to L)$ that are distributive and where D is a finite set and L is a complete lattice.

The first one includes the functions used in the alternative formulation of possibly-uninitialized variables analysis in Section 9.2 and forms the foundation of the IFDS framework presented in Section 9.4, while the second one is used in the IDE framework in Section 9.6.

The first family of functions is a special case of the second one in the following sense. The lattices $\mathcal{P}(D)$ and $D \to L$ are isomorphic (see page 43) if L is set to the two-element lattice $\{\top, \bot\}$. An element $s \in \mathcal{P}(D)$ is then represented by its characteristic function in $D \to L$:

$$m_s(d) = \begin{cases} \top & \text{if } d \in s \\ \bot & \text{if } d \notin s \end{cases}$$

Exercise 9.9: Show that another isomorphism between $\mathcal{P}(D)$ and $D' \to L$ can be obtained by setting D' to a singleton set, for example $D' = \{\star\}$, and $L = \mathcal{P}(D)$.

Let $f\colon \mathcal{P}(D) \to \mathcal{P}(D)$ be a distributive function where D is a finite set. A naive representation of f would be a table with $2^{|D|}$ entries. (If D is the set of program variables in a typical program, then such a table is huge!) A distributive function is characterized by its value on the empty set and on every singleton set, for example, $f(\{d_2, d_3\}) = f(\emptyset) \cup f(\{d_2\}) \cup f(\{d_3\})$. This means that f can be decomposed into a function $g\colon (D \cup \{\bullet\}) \to \mathcal{P}(D)$ as follows. Define

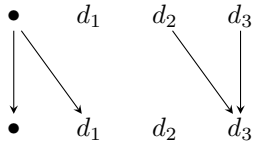
$$g(\bullet) = f(\emptyset)$$

$$g(d) = f(\{d\}) \backslash f(\emptyset) \text{ for } d \in D$$

Now $f(X) = g(\bullet) \cup \bigcup_{y \in X} g(y)$ for any $X \subseteq D$, which means that f is fully represented by g for any input. (Notice that it is safe to exclude $f(\emptyset)$ in the second case of the definition because those elements are always included due to the first case.)

Exercise 9.10: Prove that $f(X) = g(\bullet) \cup \bigcup_{y \in X} g(y)$ for any $X \subseteq D$ when f is distributive and g is defined as above.

Any such function can be represented compactly as a bipartite graph with $2 \cdot (|D| + 1)$ nodes – in other words, exponentially more succinctly than with the naive representation suggested above. As an example, with $D = \{d_1, d_2, d_3\}$, the graph



represents the function g where $g(\bullet) = \{d_1\}$, $g(d_1) = \emptyset$, and $g(d_2) = g(d_3) = \{d_3\}$, and thereby the function f where $f(S) = \{d_1, d_3\}$ if $d_2 \in S$ or $d_3 \in S$ and $f(S) = \{d_1\}$ otherwise. In general, the edges are defined by

$$\{\bullet \to \bullet\} \cup \{\bullet \to y \mid y \in f(\emptyset)\} \cup \{x \to y \mid y \in f(\{x\}) \ \wedge \ y \notin f(\emptyset)\}$$

where $x \to y$ denotes an edge from x in the top row to y in the bottom row. (The condition $y \notin f(\emptyset)$ is safe because $f(\emptyset)$ is covered by the $\bullet \to y$ edges.) Intuitively, the edges describe how the output depends on the input. In possibly-uninitialized variables analysis where $D = \mathit{Var}$, the edge $d_2 \to d_3$ means: if d_2 is possibly-uninitialized at input then d_3 is possibly-uninitialized at output. The symbol $\bullet$ can be thought of as a dataflow fact that always holds. The edge $\bullet \to d_1$ means that d_1 is unconditionally possibly-uninitialized at output.

Exercise 9.11: Assume $\mathit{Var} = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$. Draw the bipartite graph representations of the transfer functions in possibly-uninitialized variables analysis for each of the following two statements:

```
var b,d;
a = b + d;
```

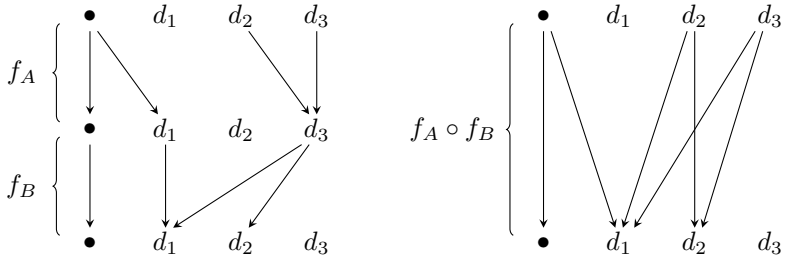
Then give a general description of how to produce the bipartite graph representations of the transfer functions for variable declarations and assignments in possibly-uninitialized variables analysis.

For this graph representation to be useful for representing jump functions, we also need two closure properties: Fortunately, distributivity is closed under both composition and least upper bound.

Exercise 9.12: Assume $f_A \colon \mathcal{P}(D) \to \mathcal{P}(D)$ and $f_B \colon \mathcal{P}(D) \to \mathcal{P}(D)$ are distributive and D is a finite set. Prove that $f_A \circ f_B$ and $f_A \sqcup f_B$ are distributive.[4]

This can be illustrated graphically with examples, here is one for composition:

[4]As usual, composition of functions is defined by $(f_A \circ f_B)(S) = f_A(f_B(S))$, and least upper bound of functions is defined by $(f_A \sqcup f_B)(S) = f_A(S) \sqcup f_B(S)$.



The edges $d_2 \to d_1$ and $d_3 \to d_1$ could be omitted in the resulting graph; they are unnecessary because of the edge $\bullet \to d_1$, but it is simpler to include them.

Exercise 9.13: Draw a similar example illustrating least upper bound of two distributive functions represented as bipartite graphs.

Exercise 9.14: Let $f_A \colon \mathcal{P}(D) \to \mathcal{P}(D)$ and $f_B \colon \mathcal{P}(D) \to \mathcal{P}(D)$ be distributive functions where D is a finite set. Describe two algorithms that, given bipartite graph representations of f_A and f_B, construct the bipartite graph representations of $f_A \circ f_B$ and $f_A \sqcup f_B$, respectively. How efficient can you make the algorithms?

To conclude this part, all the transfer functions and jump functions that appear in possibly-uninitialized variables analysis and other distributive flow-sensitive analyses with a similar analysis lattice can be represented compactly and constructed efficiently using bipartite graphs.

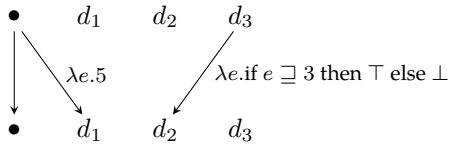
The bipartite graph representation generalizes to distributive functions of the form $f \colon (D \to L) \to (D \to L)$ where D is a finite set and L is a complete lattice. Assume f is such a function and define $g \colon (D \cup \{\bullet\}) \times (D \cup \{\bullet\}) \to (L \to L)$ as follows.

$$\begin{aligned} g(d_1, d_2)(e) &= f(\bot[d_1 \mapsto e])(d_2) \text{ for } d_1, d_2 \in D \text{ and } e \in L \\ g(\bullet, d_2)(e) &= f(\bot)(d_2) \text{ for } d_2 \in D \text{ and } e \in L \\ g(\bullet, \bullet)(e) &= e \text{ for } e \in L \\ g(d_1, \bullet)(e) &= \bot \text{ for } d_1 \in D \text{ and } e \in L \end{aligned}$$

Intuitively, $g(a, b) \colon L \to L$ is a function that specifies how the abstract value of a at input influences the abstract value of b at output. Such functions can be represented as bipartite graphs as before, but now where edges are labeled with functions of the form $L \to L$, written $d_1 \overset{m}{\to} d_2$ where $d_1, d_2 \in D \cup \{\bullet\}$ and $m \colon L \to L$. We can think of an absent edge as an edge with label $\bot$ (i.e., the bottom element of the lattice $L \to L$). It is also customary when drawing the graphs that omitting the label on an edge means the same as writing the identity function label, $\lambda e.e$. Notice that $g(\bullet, \bullet)$ is always the identity function.

As an example, assume $D = \{d_1, d_2, d_3\}$ and L is the constant propagation lattice from Section 5.2. Here is the bipartite graph representation of such a

function g:



If, for example, $m\colon D \to L$ is the function $m = [d_1 \mapsto \bot, d_2 \mapsto 42, d_3 \mapsto \top]$ then $f(m) = [d_1 \mapsto 5, d_2 \mapsto \top, d_3 \mapsto \bot]$.

Exercise 9.15: Assume $f\colon (D \to L) \to (D \to L)$ is distributive where D is a finite set and L is a complete lattice, and let g be defined from f as above. Prove that $f(m)(b) = g(\bullet, b)(\bot) \sqcup \bigsqcup_{a \in D} g(a, b)(m(a))$ for all $m\colon D \to L$ and $b \in D$. This shows that we can always use g whenever we want to compute the value of f on some input. (It also shows that the last case, $g(d_1, \bullet)(e) = \bot$, in the definition of g is in fact irrelevant.)

Exercise 9.16: The construction of g on page 122 sometimes introduces redundant edges in the bipartite graphs. In fact, it is possible to modify the first line

$$g(d_1, d_2)(e) = f(\bot[d_1 \mapsto e])(d_2) \text{ for } d_1, d_2 \in D \text{ and } e \in L$$

to return $\bot$ or e instead of $f(\bot[d_1 \mapsto e])(d_2)$ in certain cases (just like certain unnecessary edges are omitted in the unlabeled bipartite graph as discussed earlier), while preserving the correctness property from Exercise 9.15. Investigate how this can be done (see [SRH96] for inspiration).

For all this to be useful in practice, we need compact representation of all the edge functions $L \to L$ that arise for a given analysis, and we need efficient algorithms for computing composition and least upper bound of such functions. This depends on the specific analysis; an example is shown in Section 9.6.

9.4 The IFDS Framework

The ideas presented in the previous sections can be tied together with a work-list algorithm to obtain a powerful analysis framework named IFDS (Interprocedural, Finite, Distributive, Subset) [RHS95].

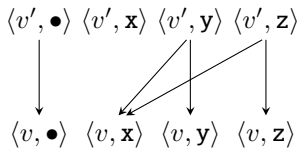
IFDS covers flow sensitive analyses where the lattice of abstract states is of the form $\mathcal{P}(D)$ for a finite set D and all transfer functions are distributive. For analysis problems that satisfy these restrictions, the framework provides full context sensitivity in polynomial time.

The elements in D represent dataflow facts, for example, "variable `x` is possibly-uninitialized". The program to be analyzed is represented as an interprocedural control flow graph (as in Section 8.1) where each node v is described by a transfer function $t_v\colon \mathcal{P}(D) \to \mathcal{P}(D)$, represented as bipartite

graphs as explained in Section 9.3. The framework is usually applied to forward analysis; backward analysis can be performed by reversing the CFG edges and swapping the roles of function entry and exit nodes. The lattice order is usually $\subseteq$, but $\supseteq$ also works.

The *exploded CFG*[5] representation of the program to be analyzed has a node for each pair $\langle v, d\rangle$ of a CFG node v and an element $d \in D \cup \{\bullet\}$. If the bipartite graph for t_{v_1} has an edge $d_1 \to d_2$ and the CFG has an edge from node v_1 to node v_2, i.e., $v_2 \in succ(v_1)$, then the exploded CFG has an edge from (v_1, d_1) to (v_2, d_2), written $\langle v_1, d_1\rangle \to \langle v_2, d_2\rangle$. Let $\mathbf{E}$ denote the set of all such edges for the given program. Parameter passing and flow of return values can be treated like assignments as in Section 9.1. The direct dataflow from call nodes to their after-call nodes for local variables that are unaffected by the function calls is modeled similarly.

As an example, if $Var = \{\mathtt{x}, \mathtt{y}, \mathtt{z}\}$ and v is an assignment `x = y + z` with a predecessor $v' \in pred(v)$, then $\mathbf{E}$ contains these edges when using the transfer function for assignments in possibly-uninitialized variables analysis:



The exploded CFG has the property that a dataflow fact $d \in D$ may hold at CFG node v if and only if $\langle v, d\rangle$ is reachable from $\langle entry_{\mathtt{main}}, \bullet\rangle$ along an interprocedurally valid path in the exploded CFG (see Exercise 9.23).

Exercise 9.17: Draw the exploded CFG for the following program using the transfer functions for possibly-uninitialized variables analysis.

```
main() {
  var a, b, c;
  b = input;
  a = plus(a, b);
  return a - c;
}

plus(d, e) {
  return d + e;
}
```

Note that D is the set $\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}, \mathtt{result}\}$ in this case.

The IFDS algorithm proceeds in two phases similar to the alternative formulation of possibly-uninitialized variables analysis in Section 9.2. The first phase (corresponding to the pre-analysis phase in Section 9.2) is called the *tabulation*

[5] Exploded CFGs are called *exploded supergraphs* in [RHS95, SRH96].

algorithm. It incrementally builds a set of *path edges* denoted $\mathbf{P}$, each written as $\langle v_1, d_1\rangle \rightsquigarrow \langle v_2, d_2\rangle$ where v_1 is a function entry node, v_2 is a CFG node in the same function as v_1, and $d_1, d_2 \in D \cup \{\bullet\}$. The idea is that the set of path edges $\{\langle v_1, d_1\rangle \rightsquigarrow \langle v_2, d_2\rangle \in \mathbf{P} \mid d_1, d_2 \in D \cup \{\bullet\}\}$ for a node v_2 constitutes the bipartite graph representation of v_2's jump function (see page 112) that relates dataflow facts at the entry v_1 of the function containing v_2 with the dataflow facts at v_2. However, reachability is handled slightly differently. Instead of tracking reachability of functions in specific contexts (as in the original formulation of possibly-uninitialized variables analysis in Section 9.1), or reachability of functions independently of contexts (as in the alternative analysis formulation in Section 9.2), reachability in IFDS is tracked at the level of individual dataflow facts: A path edge $\langle v_1, d_1\rangle \rightsquigarrow \langle v_2, d_2\rangle$ is only added to $\mathbf{P}$ if we have already established that $\langle v_1, d_1\rangle$ may be reachable from the program entry.

Phase 1 The path edges are constructed as the least solution to the following constraints for the different kinds of CFG nodes. Thus, the lattice used in this analysis phase is the powerset lattice of path edges.

The first constraint rule expresses that the program entry is always reachable:

$$\langle \mathit{entry}_{\texttt{main}}, \bullet\rangle \rightsquigarrow \langle \mathit{entry}_{\texttt{main}}, \bullet\rangle \in \mathbf{P}$$

Second, if v is a function entry node, $v_1 \in \mathit{pred}(v)$ is a call node that calls the function containing v, and v_0 is the entry node of the function containing v_1, this constraint rule models the flow of reachable dataflow facts from the call node to the function entry node:

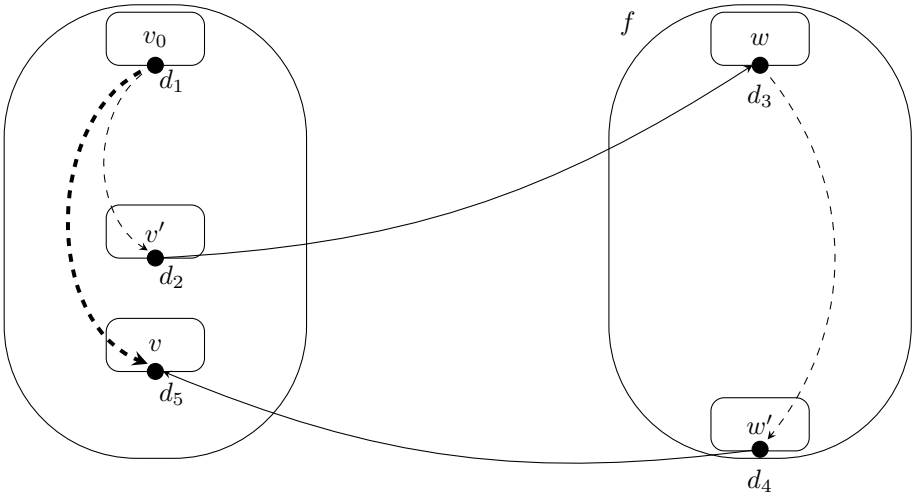
$$\forall d_1, d_2, d_3 \colon \langle v_0, d_1\rangle \rightsquigarrow \langle v_1, d_2\rangle \in \mathbf{P} \ \wedge\ \langle v_1, d_2\rangle \rightarrow \langle v, d_3\rangle \in \mathbf{E} \\ \Longrightarrow\ \langle v, d_3\rangle \rightsquigarrow \langle v, d_3\rangle \in \mathbf{P}$$

At first, this rule may look wrong, but remember that this analysis phase does not propagate dataflow facts (that is done in the second phase) but builds path edges for the nodes in the exploded CFG that are reachable from the program entry. The new edge $\langle v, d_3\rangle \rightsquigarrow \langle v, d_3\rangle$ added with this rule means that $\langle v, d_3\rangle$ may be reachable from the program entry.

Third, if v is an after-call node belonging to a call node v', v_0 is the entry node of the function that contains v and v', and furthermore, $w \in \mathit{succ}(v')$ is the entry node of the function being called with associated after-call node $w' \in \mathit{pred}(v)$, we use this constraint rule to model the flow of reachable dataflow facts from v_0 to the call node v', further through the function being called, and ending at v:

$$\forall d_1, d_2, d_3, d_4, d_5 \colon \langle v_0, d_1\rangle \rightsquigarrow \langle v', d_2\rangle \in \mathbf{P} \ \wedge\ \langle v', d_2\rangle \rightarrow \langle w, d_3\rangle \in \mathbf{E} \\ \wedge\ \langle w, d_3\rangle \rightsquigarrow \langle w', d_4\rangle \in \mathbf{P} \ \wedge\ \langle w', d_4\rangle \rightarrow \langle v, d_5\rangle \in \mathbf{E} \\ \Longrightarrow\ \langle v_0, d_1\rangle \rightsquigarrow \langle v, d_5\rangle \in \mathbf{P}$$

The dataflow involved in this rule can be illustrated as follows, where solid and dashed edges represent edges in $\mathbf{E}$ and $\mathbf{P}$, respectively, and the thick dashed edge is the new edge added by the rule:



Notice how this rule uses the path edges of the exit node w' of the function being called as a summary of the function, and that the dataflow follows interprocedurally valid paths only, which is essential for obtaining the desired context sensitivity.

The fourth constraint rule models the flow of function-local state that is transferred from a call node v' to its after-call node v, where v_0 is the entry node of the function that contains v and v':

$$\forall d_1, d_2, d_3 \colon \langle v_0, d_1\rangle \rightsquigarrow \langle v', d_2\rangle \in \mathbf{P} \ \wedge \ \langle v', d_2\rangle \to \langle v, d_3\rangle \in \mathbf{E} \implies \langle v_0, d_1\rangle \rightsquigarrow \langle v, d_3\rangle \in \mathbf{P}$$

This constraint rule also applies to any other kind of node v with a predecessor $v' \in \mathit{pred}(v)$ where v_0 is the entry node of the function that contains v and v', expressing intraprocedural dataflow that does not involve function calls.

The least solution to the constraints obtained with these rules for a given program is easily computed using a work-list algorithm (see Section 5.3).

Exercise 9.18: Explain how these constraint rules correspond to those in the pre-analysis phase in Section 9.2.

Phase 2 The second phase (corresponding to the main analysis phase in Section 9.2) that computes the final analysis results is remarkably simple and entirely intra-procedural: The abstract state $[\![v]\!] \in \mathcal{P}(D)$ for any CFG node v is obtained directly from the path edges of v, as expressed by this single rule where v_0 is the entry node in the function containing v:

$$\forall d_1, d_2 \colon \langle v_0, d_1\rangle \rightsquigarrow \langle v, d_2\rangle \in \mathbf{P} \implies d_2 \in [\![v]\!]$$

This works because a path edge $\langle v_0, d_1\rangle \rightsquigarrow \langle v, d_2\rangle \in \mathbf{P}$ has been added in the first analysis phase only if the dataflow fact d_2 may hold at v. Because of the

fine-grained tracking of reachability at the level of individual dataflow facts in the first phase, the second phase in IFDS is simpler than the main phase of the analysis in Section 9.2.

Continuing the running example, the IFDS framework can be instantiated to possibly-uninitialized variables analysis by setting $D = Var$, i.e., the set of variables in the program being analyzed, and defining the edges $\mathbf{E}$ in the exploded CFG according to the transfer functions from Section 9.1 and the control flow edges in the program CFG as explained above.

Exercise 9.19: Explain step-by-step how IFDS-based possibly-uninitialized variables analysis runs on the example programs from Exercise 9.3 and Exercise 9.17.

IFDS-based possibly-uninitialized variables analysis gives the same analysis results as the analysis from Section 9.2. Via Exercise 9.7, the IFDS formulation also has the same precision as the analysis from Section 9.1, while scalability is improved considerably, as seen in the following exercise.

Exercise 9.20: Show that the worst-case time complexity of IFDS analysis is $\mathcal{O}(|E| \cdot |D|^3)$ where E is the set of CFG edges in the program being analyzed. (Compare this result with Exercises 9.4 and 9.8.)

The precision of IFDS analysis can be stated more formally as follows. An *interprocedurally valid path* is a path $v_0 \cdot v_1 \cdot \dots \cdot v_k$ in the CFG from the program entry $v_0 = \mathit{entry}_{\texttt{main}}$ to a node $v = v_k$ where all calls and returns match, as explained earlier. Let t_{v_i} denote the transfer function of each CFG node v_i for $i = 1, 2, \dots, k$ in such a path, and let IVP denote the set of all interprocedurally valid paths. The abstract state $[\![v]\!]$ computed by IFDS for a CFG node v has the following property:

$$[\![v]\!] = \bigsqcup_{\substack{v_0 \cdot v_1 \cdot \dots \cdot v_k \in IVP \\ \text{where } v_0 = \mathit{entry}_{\texttt{main}} \text{ and } v_k = v}} (t_{v_k} \circ t_{v_{k-1}} \circ \dots \circ t_{v_2} \circ t_{v_1})(\bot)$$

This is also called the *meet-over-all-valid-paths* solution.[6]

Exercise 9.21: Argue that IFDS indeed computes meet-over-all-valid-paths solutions. (Hint: Distributivity is important for this property to hold.)

Exercise 9.22: Which of the five analyses in Exercise 5.34 fit into the IFDS framework? For the analyses that do, describe how their transfer functions can be expressed as bipartite graphs.

[6]The IFDS and IDE papers [RHS95, SRH96] use upside-down lattices compared to the presentation used here, so *join-over-all-valid-paths* would be a more appropriate name.

Exercise 9.23: As mentioned earlier in this section, the IFDS framework has an interesting connection to a certain kind of graph reachability: Dataflow fact $d \in D$ may hold at CFG node v (i.e., $d \in [\![v]\!]_2$) if and only if there exists an interprocedurally valid path from $\langle \mathit{entry}_{\texttt{main}}, \bullet \rangle$ to $\langle v, d \rangle$ in the exploded CFG. Explain why this property holds.

9.5 Copy-Constant Propagation Analysis

In constant propagation analysis from Section 5.2, the lattice of abstract states is $\mathit{Var} \to \mathit{flat}(\mathbb{Z})$. With $D = \mathit{Var}$ and $L = \mathit{flat}(\mathbb{Z})$, the transfer functions are of the form $f : (D \to L) \to (D \to L)$ that we studied in Section 9.3. Constant propagation analysis is, however, not distributive, as seen in Exercise 5.34. *Copy-constant propagation analysis* is a less precise variant that is distributive, and it is suitable for motivating the IDE framework presented in Section 9.6.

Copy-constant propagation analysis uses the same analysis lattice as ordinary constant propagation analysis, but with this less precise abstraction of operators (compare with the definition on page 57):

$$a \mathbin{\widehat{op}} b = \begin{cases} \bot & \text{if } a = \bot \text{ or } b = \bot \\ \top & \text{otherwise} \end{cases}$$

For any composite expression, such as `x*y+3`, the analysis simply yields $\top$. This means that the transfer functions for assignments, *X* = *E*, can be defined by considering only three cases of right-hand-side expressions: (1) *E* is an integer literal, *Int*, (2) *E* is a single variable, *Y* (an assignment of the form *X* = *Y* is called a copy assignment), and (3) *E* is any other expression.

Exercise 9.24: Describe the labeled bipartite graph representation of the transfer functions for the three kinds of expressions.

Exercise 9.25: Describe the result of performing context insensitive (and path insensitive) copy-constant propagation analysis on the following program.

```
main() {
  var x;
  x = p(7);
  return x;
}

p(a) {
  var y,z;
  if (a > 0) {
    y = a - 1;
    y = p(y);
  }
  z = -2 * y + 5;
  return z;
}
```

Exercise 9.26: Prove that copy-constant propagation analysis is distributive.

Exercise 9.27: Perhaps surprisingly, copy-constant propagation analysis fits into the IFDS framework. For a given program to be analyzed, let Lit be the (finite) set of integer literals that occur in the program, and choose $D = Var \times Lit$ as the set of dataflow facts. Explain how to perform copy-constant propagation analysis using IFDS with this choice of the set D.

9.6 The IDE Framework

The IDE (Interprocedural Distributive Environment) framework [SRH96] also covers flow sensitive analyses with distributive transfer functions as in IFDS, but with a larger class of abstract state lattices.

In IDE, the lattice of abstract states is $D \to L$ for some finite set D and complete lattice L with finite height.[7] As seen in Section 9.3, this generalizes the form of abstract states from IFDS. The transfer function for a CFG node v accordingly has the form $t_v \colon (D \to L) \to (D \to L)$. The notion of exploded CFGs for a given program is generalized similarly, so each edge in $\mathbf{E}$ is now labeled with a function $m \colon L \to L$, written $\langle v_1, d_1 \rangle \xrightarrow{m} \langle v_2, d_2 \rangle$. An absent edge means the same as one with label $m = \bot$ (i.e., the bottom element of $L \to L$). Intuitively, the label m expresses how the abstract value of d_1 at v_1 influences

[7] An element of the lattice $D \to L$ is called an *environment*, which explains the 'E' in IDE.

the abstract value of d_2 at v_2. (In comparison, in IFDS the presence or absence of an edge $\langle v_1, d_1\rangle \to \langle v_2, d_2\rangle$ expresses whether or not d_1 at v_1 influences d_2 at v_2.)

The edges in $\mathbf{E}$ can be constructed from the program being analyzed using the bipartite graph representation from Section 9.3: If the bipartite graph for $t_{v_1} : (D \to L) \to (D \to L)$ has an edge $d_1 \overset{m}{\to} d_2$ and the CFG has an edge from node v_1 to node v_2, i.e., $v_2 \in succ(v_1)$, then the exploded CFG has a corresponding edge $\langle v_1, d_1\rangle \overset{m}{\to} \langle v_2, d_2\rangle \in \mathbf{E}$.

Exercise 9.28: Draw the exploded CFG for the program from Exercise 9.25 using the transfer functions from copy-constant propagation analysis.

In IDE, path edges (called jump functions in [SRH96]) are similarly labeled with functions of the form $L \to L$. Recall from Section 9.4 that IFDS builds a set of path edges $\mathbf{P}$, starting with the empty set of edges and in each step adding an edge. IDE instead builds the edge labels, starting with $\bot$ everywhere and in each step moving upwards in the $L \to L$ lattice at some edge. For this reason, we introduce a different notation. For each pair of exploded CFG nodes, $\langle v_1, d_1\rangle$ and $\langle v_2, d_2\rangle$, the constraint variable $[\![\langle v_1, d_1\rangle \rightsquigarrow \langle v_2, d_2\rangle]\!]_{\mathbf{P}} : L \to L$ denotes the label of the corresponding edge. We similarly use the notation $[\langle v_1, d_1\rangle \to \langle v_2, d_2\rangle]_{\mathbf{E}}$ for the label of the edge from $\langle v_1, d_1\rangle$ to $\langle v_2, d_2\rangle$ in $\mathbf{E}$.

Phase 1 The first phase of IDE builds the path edges, much like phase one in IFDS but using the more general lattice consisting of $L \to L$ functions. Each of the following constraint rules directly corresponds to one of the constraint rules in IFDS.

The first constraint rule expresses that the program entry is always reachable and has the identity jump function:

$$id \sqsubseteq [\![\langle \mathit{entry}_{\texttt{main}}, \bullet\rangle \rightsquigarrow \langle \mathit{entry}_{\texttt{main}}, \bullet\rangle]\!]_{\mathbf{P}}$$

where $id : L \to L$ is the identity function, $id = \lambda e.e$ for all $e \in L$.

Second, if v is a function entry node, $v_1 \in pred(v)$ is a call node that calls the function containing v, and v_0 is the entry node of the function containing v_1, this constraint rule models reachability of exploded CFG nodes at function entries:

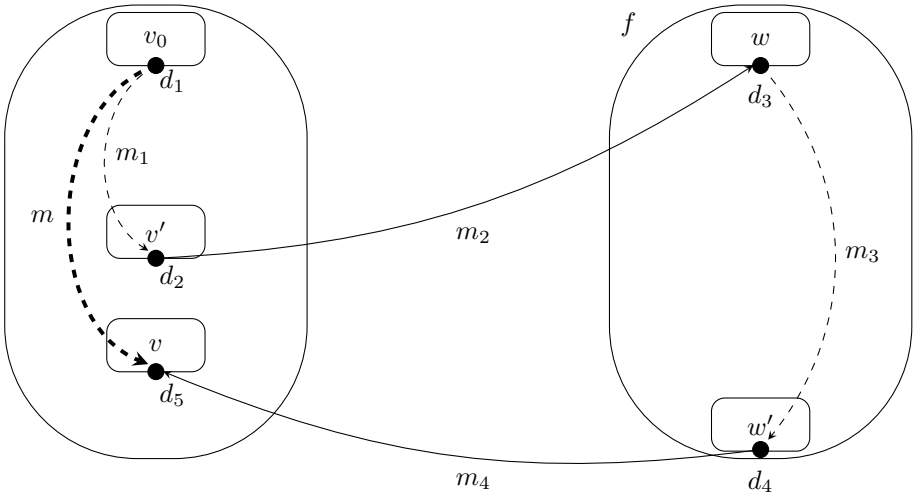
$$\forall d_1, d_2, d_3 : [\![\langle v_0, d_1\rangle \rightsquigarrow \langle v_1, d_2\rangle]\!]_{\mathbf{P}} \neq \bot \ \wedge\ [\langle v_1, d_2\rangle \to \langle v, d_3\rangle]_{\mathbf{E}} \neq \bot \implies id \sqsubseteq [\![\langle v, d_3\rangle \rightsquigarrow \langle v, d_3\rangle]\!]_{\mathbf{P}}$$

Third, if v is an after-call node belonging to a call node v', v_0 is the entry node of the function that contains v and v', and furthermore, $w \in succ(v')$ is the entry node of the function being called with associated after-call node $w' \in pred(v)$, the following constraint rule models the dataflow from v_0 to the call node v',

further through the function being called, and ending at v:

$$\begin{aligned}&\forall d_1, d_2, d_3, d_4, d_5 : \\ &\quad m_1 = [\![\langle v_0, d_1\rangle \rightsquigarrow \langle v', d_2\rangle]\!]_{\mathbf{P}} \neq \bot \ \wedge\ m_2 = [\langle v', d_2\rangle \rightarrow \langle w, d_3\rangle]_{\mathbf{E}} \neq \bot \\ &\quad \wedge\ m_3 = [\![\langle w, d_3\rangle \rightsquigarrow \langle w', d_4\rangle]\!]_{\mathbf{P}} \neq \bot \ \wedge\ m_4 = [\langle w', d_4\rangle \rightarrow \langle v, d_5\rangle]_{\mathbf{E}} \neq \bot \\ &\qquad\qquad \Longrightarrow\ m_4 \circ m_3 \circ m_2 \circ m_1 \sqsubseteq [\![\langle v_0, d_1\rangle \rightsquigarrow \langle v, d_5\rangle]\!]_{\mathbf{P}}\end{aligned}$$

The composition of the edge functions expresses how the abstract value of $\langle v_0, d_1\rangle$ influences the abstract value of $\langle v, d_5\rangle$. This rule can be illustrated like the corresponding rule in IFDS but now with edge labels (where $m = m_4 \circ m_3 \circ m_2 \circ m_1$):



The fourth constraint rule applies to after-call nodes v where v' is the associated call node and v_0 is the entry node of the function that contains v and v':

$$\begin{aligned}&\forall d_1, d_2, d_3 : m_1 = [\![\langle v_0, d_1\rangle \rightsquigarrow \langle v', d_2\rangle]\!]_{\mathbf{P}} \neq \bot \ \wedge\ m_2 = [\langle v', d_2\rangle \rightarrow \langle v, d_3\rangle]_{\mathbf{E}} \neq \bot \\ &\qquad\qquad \Longrightarrow\ m_2 \circ m_1 \sqsubseteq [\![\langle v_0, d_1\rangle \rightsquigarrow \langle v, d_3\rangle]\!]_{\mathbf{P}}\end{aligned}$$

Similar to IFDS, this constraint rule also applies to any other kind of node v with a predecessor $v' \in \mathit{pred}(v)$ where v_0 is the entry node of the function that contains v and v'.

Phase 2 The second phase of IDE analysis uses the path edges (representing jump functions) that have been produced in the first phase to compute an abstract value $[\![\langle v, d\rangle]\!] \in \mathit{lift}(L)$ for each exploded CFG node $\langle v, d\rangle$. We here use a lifted lattice with the bottom element unreachable representing nodes that are unreachable from the program entry. This analysis phase closely resembles the main phase of the analysis described in Section 9.2.

As usual, we specify the analysis in a constraint-based style. First, the program entry is always reachable:

$$\forall d\colon [\![\langle \mathit{entry}_{\texttt{main}}, d\rangle]\!] \neq \mathsf{unreachable}$$

Second, at any CFG node v, the abstract value of $d \in D$ can be obtained from the abstract values at the entry v_0 of the function containing v and the path edges that lead from v_0 to v:

$$\forall d_0, d\colon [\![\langle v_0, d_0\rangle]\!] \neq \mathsf{unreachable} \ \wedge \ m = [\![\langle v_0, d_0\rangle \rightsquigarrow \langle v, d\rangle]\!]_{\mathbf{P}} \\ \implies m([\![\langle v_0, d_0\rangle]\!]) \sqsubseteq [\![\langle v, d\rangle]\!]$$

Third, for each function entry node v where $v_1 \in \mathit{pred}(v)$ is a call node that calls the function containing v, we model the dataflow from v_1 to v (typically parameter passing) by propagating abstract values according to the edges in the exploded CFG from v_1 to v:

$$\forall d_1, d\colon [\![\langle v_1, d_1\rangle]\!] \neq \mathsf{unreachable} \ \wedge \ m = [\langle v_1, d_1\rangle \rightarrow \langle v, d\rangle]_{\mathbf{E}} \\ \implies m([\![\langle v_1, d_1\rangle]\!]) \sqsubseteq [\![\langle v, d\rangle]\!]$$

Finally, the resulting abstract state for each CFG node v, denoted $[\![v]\!]_2\colon D \to L$, is defined by $[\![v]\!]_2(d) = [\![\langle v, d\rangle]\!] \in L$ for $d \in D$.

The least solution to the resulting constraints for each analysis phase for a given program can be computed using a work-list algorithm, much like for IFDS but with a little more effort for the second phase (pseudo-code can be found in [SRH96]).

Exercise 9.29: Explain why the second phase of IDE does not need separate constraints for modeling the flow of return values from function exit nodes to after-call nodes.

The efficiency of IDE depends on the edge label functions being efficiently representable. This means that functions that appear on the edges $\mathbf{E}$ in the exploded CFG and on the path edges $\mathbf{P}$ can be represented in constant space (i.e., independently of the size of the program), and function composition, least upper bound, equality, and function application can be computed in constant time. For copy-constant propagation analysis, the edge labels in $\mathbf{E}$ are identity functions and constant functions. The functions obtained from these using composition and least upper bound can also be represented efficiently, as shown in the following exercise.

Exercise 9.30: Let F denote the set of functions

$$F = \{\lambda e.c \mid c \in L\} \cup \{\lambda e.e \sqcup c \mid c \in L\},$$

where L is a complete lattice. Show that F is closed under function composition and least upper bound. Notice that the identity function belongs to F by taking $c = \bot$.

Exercise 9.31: Show step-by-step how IDE-based copy-constant propagation analysis runs on the example program from Exercise 9.28. Is the analysis result more precise than the context insensitive analysis?

As discussed in Section 9.4, the IFDS algorithm computes the meet-over-all-valid paths solutions in polynomial time (while an analysis based directly on the functional approach to context sensitivity as the one presented in Section 8.4 computes the same results but in worst-case exponential time). The IDE approach similarly has two key properties:

1. The IDE algorithm also produces meet-over-all-valid paths solutions (but for the more general class of analysis problems than IFDS).
2. The worst-case time complexity of IDE analysis is the same as for IFDS: $\mathcal{O}(|E|\cdot|D|^3)$ where E is the set of CFG edges in the program being analyzed, provided that the edge functions are efficiently representable and the height of L is constant.

Exercise 9.32: Argue that IDE computes meet-over-all-valid-paths solutions. (See Exercise 9.21.)

Exercise 9.33: Prove the statement above about the worst-case time complexity of IDE analysis.

Interestingly, the IDE framework is sometimes preferable also for analysis problems that fit into IFDS. Consider, for example, copy-constant propagation analysis as discussed in Exercise 9.27. With IFDS, this analysis requires a set of dataflow facts of size $|Var| \cdot |Lits|$, whereas the size of $Lits$ is irrelevant in the IDE version. Depending on implementation techniques, this may have a large effect on the analysis performance.

Exercise 9.34: One approach to compute possibly-uninitialized variables with IDE is to use the characteristic function isomorphism discussed in Section 9.3. Another approach is to use the alternative isomorphism from Exercise 9.9. Which of those two approaches is preferable regarding analysis scalability?

Chapter 10

Control Flow Analysis

If we introduce functions as values (and thereby higher-order functions), then at each call site, it is no longer trivial to see which function is being called. A similar challenge arises in object-oriented languages with methods that are invoked using dynamic dispatching. The task of *control flow analysis* is to conservatively approximate the interprocedural control flow, also called the *call graph*, for such programs. A call graph shows for each call site which functions may be called, expressed as *call edges* from the AST node or CFG node representing the call site to the functions that may be called. We usually aim for an over-approximation, meaning that call graphs may contain too many call edges but never too few. Thereby, the call graph provides a foundation for subsequently performing interprocedural dataflow analysis, such as constant propagation analysis.

10.1 Closure Analysis for the λ-calculus

Control flow analysis in its purest form can best be illustrated by the classical λ-calculus. Its abstract syntax is defined by this grammar:

$$\begin{aligned} Exp \rightarrow\ & \lambda Id.Exp \\ \mid\ & Id \\ \mid\ & Exp\ Exp \end{aligned}$$

(In Section 10.3 we demonstrate this analysis technique on the TIP language.) The concrete syntax also allows parentheses.[1] 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

[1] As customary, application is left-associative, and application has higher precedence than abstraction.

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.

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

$$\lambda X \in [\![E_1]\!] \implies ([\![E_2]\!] \subseteq [\![X]\!] \ \wedge\ [\![E]\!] \subseteq [\![E_1\ E_2]\!])$$

for every abstraction $\lambda X.E$, which models that the actual argument E_2 may flow into the formal argument X and that the value of the function body E is among the possible results of the function call $E_1\ E_2$.

As an example, for the small program $(\lambda s.\lambda z.sz)(\lambda n.\lambda t.\lambda e.e)(\lambda r.\lambda p.r)$, which contains 7 λ-abstractions and 3 applications, the following constraints are produced:

$$\begin{array}{l}
\lambda s \in [\![\lambda s.\lambda z.sz]\!] \\
\lambda z \in [\![\lambda z.sz]\!] \\
\lambda n \in [\![\lambda n.\lambda t.\lambda e.e]\!] \\
\lambda t \in [\![\lambda t.\lambda e.e]\!] \\
\lambda e \in [\![\lambda e.e]\!] \\
\lambda r \in [\![\lambda r.\lambda p.r]\!] \\
\lambda p \in [\![\lambda p.r]\!] \\
\lambda s \in [\![\lambda s.\lambda z.sz]\!] \implies \\
\quad ([\![\lambda n.\lambda t.\lambda e.e]\!] \subseteq [\![s]\!] \ \wedge\ [\![\lambda z.sz]\!] \subseteq [\![(\lambda s.\lambda z.sz)(\lambda n.\lambda t.\lambda e.e)]\!]) \\
\lambda z \in [\![\lambda s.\lambda z.sz]\!] \implies \\
\quad ([\![\lambda n.\lambda t.\lambda e.e]\!] \subseteq [\![z]\!] \ \wedge\ [\![sz]\!] \subseteq [\![(\lambda s.\lambda z.sz)(\lambda n.\lambda t.\lambda e.e)]\!]) \\
\lambda n \in [\![\lambda s.\lambda z.sz]\!] \implies \\
\quad ([\![\lambda n.\lambda t.\lambda e.e]\!] \subseteq [\![n]\!] \ \wedge\ [\![\lambda t.\lambda e.e]\!] \subseteq [\![(\lambda s.\lambda z.sz)(\lambda n.\lambda t.\lambda e.e)]\!]) \\
\lambda t \in [\![\lambda s.\lambda z.sz]\!] \implies \\
\quad ([\![\lambda n.\lambda t.\lambda e.e]\!] \subseteq [\![t]\!] \ \wedge\ [\![\lambda e.e]\!] \subseteq [\![(\lambda s.\lambda z.sz)(\lambda n.\lambda t.\lambda e.e)]\!]) \\
\lambda e \in [\![\lambda s.\lambda z.sz]\!] \implies \\
\quad ([\![\lambda n.\lambda t.\lambda e.e]\!] \subseteq [\![e]\!] \ \wedge\ [\![e]\!] \subseteq [\![(\lambda s.\lambda z.sz)(\lambda n.\lambda t.\lambda e.e)]\!]) \\
\lambda r \in [\![\lambda s.\lambda z.sz]\!] \implies \\
\quad ([\![\lambda n.\lambda t.\lambda e.e]\!] \subseteq [\![r]\!] \ \wedge\ [\![\lambda p.r]\!] \subseteq [\![(\lambda s.\lambda z.sz)(\lambda n.\lambda t.\lambda e.e)]\!]) \\
\lambda p \in [\![\lambda s.\lambda z.sz]\!] \implies \\
\quad ([\![\lambda n.\lambda t.\lambda e.e]\!] \subseteq [\![p]\!] \ \wedge\ [\![r]\!] \subseteq [\![(\lambda s.\lambda z.sz)(\lambda n.\lambda t.\lambda e.e)]\!]) \\
\lambda s \in [\![s]\!] \implies \\
\quad ([\![z]\!] \subseteq [\![s]\!] \ \wedge\ [\![\lambda z.sz]\!] \subseteq [\![sz]\!]) \\
\lambda z \in [\![s]\!] \implies \\
\quad ([\![z]\!] \subseteq [\![z]\!] \ \wedge\ [\![sz]\!] \subseteq [\![sz]\!]) \\
\lambda n \in [\![s]\!] \implies
\end{array}$$

$$
\begin{array}{l}
\quad (\llbracket z \rrbracket \subseteq \llbracket n \rrbracket \ \wedge \ \llbracket \lambda t.\lambda e.e \rrbracket \subseteq \llbracket sz \rrbracket) \\
\lambda t \in \llbracket s \rrbracket \implies \\
\quad (\llbracket z \rrbracket \subseteq \llbracket t \rrbracket \ \wedge \ \llbracket \lambda e.e \rrbracket \subseteq \llbracket sz \rrbracket) \\
\lambda e \in \llbracket s \rrbracket \implies \\
\quad (\llbracket z \rrbracket \subseteq \llbracket e \rrbracket \ \wedge \ \llbracket e \rrbracket \subseteq \llbracket sz \rrbracket) \\
\lambda r \in \llbracket s \rrbracket \implies \\
\quad (\llbracket z \rrbracket \subseteq \llbracket r \rrbracket \ \wedge \ \llbracket \lambda p.r \rrbracket \subseteq \llbracket sz \rrbracket) \\
\lambda p \in \llbracket s \rrbracket \implies \\
\quad (\llbracket z \rrbracket \subseteq \llbracket p \rrbracket \ \wedge \ \llbracket r \rrbracket \subseteq \llbracket sz \rrbracket) \\
\lambda s \in \llbracket (\lambda s.\lambda z.sz)(\lambda n.\lambda t.\lambda e.e) \rrbracket \implies \\
\quad (\llbracket \lambda r.\lambda p.r \rrbracket \subseteq \llbracket s \rrbracket \ \wedge \ \llbracket \lambda z.sz \rrbracket \subseteq \llbracket (\lambda s.\lambda z.sz)(\lambda n.\lambda t.\lambda e.e)(\lambda r.\lambda p.r) \rrbracket) \\
\lambda z \in \llbracket (\lambda s.\lambda z.sz)(\lambda n.\lambda t.\lambda e.e) \rrbracket \implies \\
\quad (\llbracket \lambda r.\lambda p.r \rrbracket \subseteq \llbracket z \rrbracket \ \wedge \ \llbracket sz \rrbracket \subseteq \llbracket (\lambda s.\lambda z.sz)(\lambda n.\lambda t.\lambda e.e)(\lambda r.\lambda p.r) \rrbracket) \\
\lambda n \in \llbracket (\lambda s.\lambda z.sz)(\lambda n.\lambda t.\lambda e.e) \rrbracket \implies \\
\quad (\llbracket \lambda r.\lambda p.r \rrbracket \subseteq \llbracket n \rrbracket \ \wedge \ \llbracket \lambda t.\lambda e.e \rrbracket \subseteq \llbracket (\lambda s.\lambda z.sz)(\lambda n.\lambda t.\lambda e.e)(\lambda r.\lambda p.r) \rrbracket) \\
\lambda t \in \llbracket (\lambda s.\lambda z.sz)(\lambda n.\lambda t.\lambda e.e) \rrbracket \implies \\
\quad (\llbracket \lambda r.\lambda p.r \rrbracket \subseteq \llbracket t \rrbracket \ \wedge \ \llbracket \lambda e.e \rrbracket \subseteq \llbracket (\lambda s.\lambda z.sz)(\lambda n.\lambda t.\lambda e.e)(\lambda r.\lambda p.r) \rrbracket) \\
\lambda e \in \llbracket (\lambda s.\lambda z.sz)(\lambda n.\lambda t.\lambda e.e) \rrbracket \implies \\
\quad (\llbracket \lambda r.\lambda p.r \rrbracket \subseteq \llbracket e \rrbracket \ \wedge \ \llbracket e \rrbracket \subseteq \llbracket (\lambda s.\lambda z.sz)(\lambda n.\lambda t.\lambda e.e)(\lambda r.\lambda p.r) \rrbracket) \\
\lambda r \in \llbracket (\lambda s.\lambda z.sz)(\lambda n.\lambda t.\lambda e.e) \rrbracket \implies \\
\quad (\llbracket \lambda r.\lambda p.r \rrbracket \subseteq \llbracket r \rrbracket \ \wedge \ \llbracket \lambda p.r \rrbracket \subseteq \llbracket (\lambda s.\lambda z.sz)(\lambda n.\lambda t.\lambda e.e)(\lambda r.\lambda p.r) \rrbracket) \\
\lambda p \in \llbracket (\lambda s.\lambda z.sz)(\lambda n.\lambda t.\lambda e.e) \rrbracket \implies \\
\quad (\llbracket \lambda r.\lambda p.r \rrbracket \subseteq \llbracket p \rrbracket \ \wedge \ \llbracket r \rrbracket \subseteq \llbracket (\lambda s.\lambda z.sz)(\lambda n.\lambda t.\lambda e.e)(\lambda r.\lambda p.r) \rrbracket)
\end{array}
$$

Note that constraints that contain conjunctions can be rewritten as multiple individual constraints. For example, $\lambda z \in \llbracket s \rrbracket \implies (\llbracket z \rrbracket \subseteq \llbracket z \rrbracket \ \wedge \ \llbracket sz \rrbracket \subseteq \llbracket sz \rrbracket)$ can be rewritten as the two constraints $\lambda z \in \llbracket s \rrbracket \implies \llbracket z \rrbracket \subseteq \llbracket z \rrbracket$ and $\lambda z \in \llbracket s \rrbracket \implies \llbracket sz \rrbracket \subseteq \llbracket sz \rrbracket$.

We shall see in the following section how to compute solutions to such a collection of constraints. For this specific collection of constraints in this example, the least solution is as follows.

$$
\begin{array}{l}
\llbracket s \rrbracket = \{\lambda n\} \\
\llbracket z \rrbracket = \{\lambda r\} \\
\llbracket n \rrbracket = \{\lambda r\} \\
\llbracket t \rrbracket = \emptyset \\
\llbracket e \rrbracket = \emptyset \\
\llbracket r \rrbracket = \emptyset \\
\llbracket p \rrbracket = \emptyset \\
\llbracket \lambda s.\lambda z.sz \rrbracket = \{\lambda s\} \\
\llbracket \lambda z.sz \rrbracket = \{\lambda z\} \\
\llbracket sz \rrbracket = \{\lambda t\} \\
\llbracket \lambda n.\lambda t.\lambda e.e \rrbracket = \{\lambda n\} \\
\llbracket \lambda t.\lambda e.e \rrbracket = \{\lambda t\} \\
\llbracket \lambda e.e \rrbracket = \{\lambda e\} \\
\llbracket \lambda r.\lambda p.r \rrbracket = \{\lambda r\} \\
\llbracket \lambda p.r \rrbracket = \{\lambda p\}
\end{array}
$$

$[\![(\lambda s.\lambda z.sz)(\lambda n.\lambda t.\lambda e.e)]\!] = \{\lambda z\}$
$[\![(\lambda s.\lambda z.sz)(\lambda n.\lambda t.\lambda e.e)(\lambda r.\lambda p.r)]\!] = \{\lambda t\}$

In particular, we see from this solution that s in the application sz can only be the abstraction $\lambda n.\lambda t.\lambda e.e$ (according to the solution for $[\![s]\!]$), and at the outermost application (corresponding to the entire program), the abstraction being applied can only be $\lambda z.sz$ (according to the solution for $[\![(\lambda s.\lambda z.sz)(\lambda n.\lambda t.\lambda e.e)]\!]$).

Exercise 10.1: Show that the resulting constraints for any λ-calculus term are monotone and can be solved by a fixed-point computation, with an appropriate choice of lattice.

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

Exercise 10.2: Show that a unique least solution exists, since solutions are closed under intersection.

The following algorithm can find the least solution to the collection of constraints in time $\mathcal{O}(n^3)$ for a program of size n (for example, measured by the number of AST nodes). It maintains a directed graph where nodes correspond to constraint variables and edges reflect inclusion constraints. For each constraint variable x,

- $x.sol \subseteq T$ holds the solution for x,
- $x.succ \subseteq V$ is the set of successors of x (i.e., the edges of the graph), and
- $x.cond(t) \subseteq V \times V$ represents a set of conditional constraints for x and t.

Additionally we have

- $W \subseteq T \times V$ is a worklist.

All the sets are initially empty. The constraints are processed one by one using the following helper functions for adding tokens and edges and propagating tokens to satisfy the constraints that have been seen so far.

```
procedure addToken(t, x)
    if t ∉ x.sol then
        x.sol.add(t)
        W.add(t, x)
```

```
    end if
end procedure

procedure addEdge(x, y)
    if x ≠ y ∧ y ∉ x.succ then
        x.succ.add(y)
        for t in x.sol do
            addToken(t, y)
        end for
    end if
end procedure

procedure propagate()
    while W ≠ ∅ do
        (t, x) := W.removeNext()
        for (y, z) in x.cond(t) do
            addEdge(y, z)
        end for
        for y in x.succ do
            addToken(t, y)
        end for
    end while
end procedure
```

The constraints are now processed as follows.

$t \in x$**:**

```
addToken(t, x)
propagate()
```

$t \in x \implies y \subseteq z$**:**

```
if t ∈ x.sol then
    addEdge(y, z)
    propagate()
else
    x.cond(t).add(y, z)
end if
```

Exercise 10.3: Show, step by step, how this algorithm finds the smallest solution for the constraints from the example program in Section 10.1.

Exercise 10.4: Explain how the algorithm can be extended to also support unconditional inclusion constraints of the form $x \subseteq y$.

To analyze the time complexity of 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. The number of constraints generated is $\mathcal{O}(n)$ for the first kind and $\mathcal{O}(n^2)$ for the second kind.

Each pair of a token and a variable can be added at most once to the worklist, so the number of iterations of PROPAGATE is $\mathcal{O}(n^2)$. For the same reason, W can be implemented with constant time operations using an array. The ADDEDGE function is called $\mathcal{O}(n^2)$ times in total because there are $\mathcal{O}(n^2)$ conditional constraints, and each of them is processed either immediately or at most once by PROPAGATE. The function ADDTOKEN takes time $\mathcal{O}(1)$ if representing each $x.sol$ as a bit vector. The ADDEDGE function calls ADDTOKEN $\mathcal{O}(n^3)$ times in total because there are $\mathcal{O}(n^2)$ edges and $\mathcal{O}(n)$ tokens. If representing $x.succ$ using, for example, arrays or hashtables, membership testing and insertion take time $\mathcal{O}(n)$ in the worst case. This means that all calls to ADDEDGE take time $\mathcal{O}(n^3)$ in total. Each iteration of PROPAGATE takes time $\mathcal{O}(n)$ if excluding the time for ADDEDGE, assuming that $x.cond(t)$ is also implemented using arrays or hashtables (and duplicates in $x.cond(t)$ do not affect correctness). Adding up, the total cost for the algorithm is $\mathcal{O}(n^3)$. The fact that this seems like a lower bound as well is referred to as the *cubic time bottleneck*.[2]

Numerous algorithmic variations and improvements are possible, including:

- cycle elimination (collapsing nodes if there is a cycle of inclusion constraints) [FFSA98],
- maintaining the worklist in topological order [PKH07],
- interleaving solution propagation and constraint processing [PKH07, PB09],
- using a shared bit vector representation to save memory [HT01b, SCD+13],
- type filtering [LH03, SCD+13],
- on-demand processing [HT01a],
- difference propagation [LH03, PKH04], and
- subsumed node compaction [RC00].

These techniques have mostly been studied in connection with points-to analysis (see Chapter 11). For a control flow analysis perspective, see the survey by Midtgaard [Mid12].

10.3 TIP with First-Class Functions

Consider now our tiny language TIP where we allow functions as values. For a computed function call $E(E_1, \ldots, E_n)$ we cannot see directly from the syntax

[2] It is possible to improve this slightly to $\mathcal{O}(n^3/\log n)$ using a bit vector representation assuming a machine with word size $\Theta(\log n)$ [Cha08].

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.

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:

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

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.

We can optionally also introduce a simpler rule for the special case where the function being called is given directly by name (as in simple function calls before we added first-class functions to TIP). For a direct function call $f(E_1, \ldots, E_n)$ where f is a function with arguments $a_f^1, \ldots, a_f^n$ and return expression E'_f, we have this (unconditional) constraint:

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

With the result of a control flow analysis, we can construct a CFG as in Section 8.1 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;
}
```

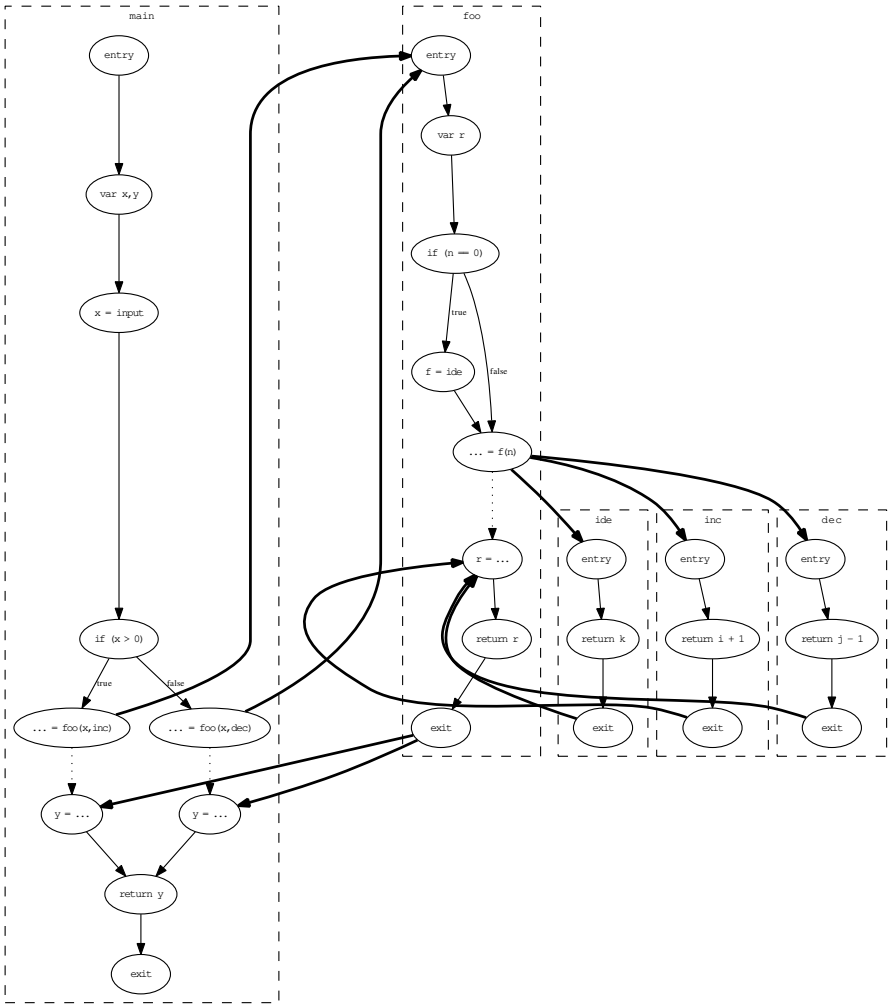
The control flow analysis (with the special rule for direct calls) generates the following constraints:

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

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 interprocedural CFG for the program, which then can be used as a basis for subsequent interprocedural dataflow analyses.


main
entry
var x,y
x = input
if (x > 0)
true
false
... = foo(x,inc)
... = foo(x,dec)
y = ...
y = ...
return y
exit
foo
entry
var r
if (n == 0)
true
false
f = ide
... = f(n)
r = ...
return r
exit
ide
entry
return k
exit
inc
entry
return i + 1
exit
dec
entry
return j - 1
exit

Exercise 10.5: 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) Compute the least solution to the constraints.

Exercise 10.6: As an alternative to the analysis described above, design a flow-sensitive control flow analysis as a monotone framework (i.e., in the style of Chapters 5 and 8). (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;
}
```

10.4 Control Flow in Object Oriented Languages

A language with functions as values must use the kind of control flow analysis described in the previous sections to obtain a reasonably precise CFG. For com-

mon 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 11

Pointer Analysis

The last TIP language feature we consider is pointers and dynamic memory allocation (see the syntax in Section 2.1). For simplicity, we ignore records (except in the last part of Section 11.2).

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

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

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.

11.1 Allocation-Site Abstraction

We first focus on intraprocedural analysis and postpone treatment of function calls to Section 11.4.

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* [CWZ90], 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 corresponding source location, hence the name allocation-site abstraction. We use Cell to denote the set of abstract cells for the given program.

The first analyses that we shall study are flow-insensitive. The end result of such an analysis is a function $pt\colon Var \to \mathcal{P}(Cell)$ that for each pointer variable X returns the set $pt(X)$ of abstract cells it may point to. This class of analyses is called *points-to analysis*. 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 local variables are undefined in TIP programs; however, as for the type analysis in Chapter 3 we assume that all the variables are initialized before they are used. (In other words, these analyses are sound only for programs that never read from uninitialized variables; see also Section 5.9.)

An almost-trivial analysis, called *address taken*, is to simply return all possible abstract cells, except that any variable 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 cells whose types do not match the type of the pointer variable.

11.2 Andersen's Algorithm

One approach to points-to analysis, called Andersen's algorithm [And94] or *inclusion-based* points-to analysis, is quite similar to control flow analysis. For each abstract cell c we introduce a constraint variable $[\![c]\!]$ ranging over sets of abstract cells.

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

- X `= alloc` P`;` where P is `null` or an integer constant
- X_1 `= &`X_2`;`
- X_1 `=` X_2`;`
- X_1 `= *`X_2`;`
- `*`X_1 `=` X_2`;`
- X `= null;`

Exercise 11.1: Explain how this normalization can be performed systematically by introducing fresh temporary variables. (See also Exercise 2.4.)

For each of these pointer operations we then generate constraints:

X = `alloc` P: `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: $c \in [\![X_2]\!] \implies [\![c]\!] \subseteq [\![X_1]\!]$ for each $c \in Cell$
`*`$X_1 = X_2$: $c \in [\![X_1]\!] \implies [\![X_2]\!] \subseteq [\![c]\!]$ for each $c \in Cell$

The last two constraints are generated for every abstract cell $c \in Cell$, but it is safe to skip the abstract cells that represent program variables X where `&`X does not occur in the program. 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 10.2. The resulting points-to function is defined simply as $pt(p) = [\![p]\!]$.

Consider the following example program fragment.

```
p = alloc null;
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`$]\!]$
$c \in [\![$`p`$]\!] \implies [\![$`z`$]\!] \subseteq [\![c]\!]$ for each $c \in Cell$
$[\![$`q`$]\!] \subseteq [\![$`p`$]\!]$
`y` $\in [\![$`q`$]\!]$
$c \in [\![$`p`$]\!] \implies [\![c]\!] \subseteq [\![$`x`$]\!]$ for each $c \in Cell$
`z` $\in [\![$`p`$]\!]$

where $Cell = \{$`p`, `q`, `x`, `y`, `z`, `alloc-1`$\}$. The points-to sets for the least solution are quite precise in this case:

$pt($`p`$) = \{$`alloc-1`, `y`, `z`$\}$
$pt($`q`$) = \{$`y`$\}$
$pt($`x`$) = pt($`y`$) = pt($`z`$) = \emptyset$

Note that although this algorithm is flow insensitive, the directionality of the set inclusion constraints implies that the dataflow is still modeled with some accuracy.

Exercise 11.2: 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 null;
```

Exercise 11.3: 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;
}
```

The following two exercises show that Andersen's analysis is, perhaps surprisingly, not the most precise possible flow-insensitive points-to analysis. This is mostly of theoretical interest, however [BCSS11]. The lack of flow-sensitivity (see Section 11.6) and path-sensitivity (and context sensitivity for interprocedural analysis, see Section 11.4) are much more important factors in practice.

Exercise 11.4: Use Andersen's algorithm to compute the points-to sets for the variables in the following program fragment:

```
a = &b;
a = &x;
b = &c;
c = &d;
x = &y;
y = &z;
*a = **a;
```

Notice that the last statement has not been normalized. As observed by Horwitz [Hor97], the normalization of the statement causes imprecision in the following sense: According to Andersen's analysis, which requires the statement to be normalized using two temporary variables, `b` and `x` may be aliases, but that is not possible with any TIP program that contains the above set of assignments, no matter which control flow exists between them.

Exercise 11.5: This example is due to Chakaravarthy [Cha03]:

```
a = &c;
c = &b;
b = &a;
b = a;
*b = c;
d = *a;
```

(a) Show that Andersen's analysis concludes for this code that `d` may point to `c`.

(b) Argue that for any TIP program that has this set of assignments, no matter which control flow exists between them, `d` never points to `c` in any execution. (Hint: `a` may point to `b`, and `b` may point to `c`, but not simultaneously.)

Exercise 11.6: Recall from Exercise 5.26 that an analysis is distributive if all its constraint functions are distributive. Show that Andersen's analysis is *not* distributive. (Hint: consider the constraint for the statement `x=*y` or `*x=y`.)

Let us now consider how to perform pointer analysis for TIP programs that also use records (see Section 2.1 and Exercise 5.10). One simple approach called *field insensitive* analysis is to treat all fields in a record as equivalent, which is trivial for Andersen's analysis. Another simple approach called *field-based* analysis is to conversely treat each field name as a single global variable without distinguishing between the different cells that contain the fields. Those

approaches are sometimes too imprecise, so instead we describe in more detail a *field-sensitive* extension of Andersen's analysis tailored for TIP. We impose one restriction, however: for this analysis, the values of record fields cannot themselves be records. (The values of record fields can be pointers to records, so this is not a severe restriction in practice; see Exercise 11.8.)

As a first step, we allocate constraint variables also for all possible fields in all cells, so we now have $[\![\cdot]\!]\colon \big(\mathit{Cell} \cup (\mathit{Cell} \times \mathit{Field})\big) \to \mathcal{P}(\mathit{Cell})$ where Field is the set of field names that occur in the program being analyzed. For convenience we use the notation $[\![c\,\texttt{.}\,f]\!]$ instead of $[\![(c,f)]\!]$ for $c \in \mathit{Cell}$, $f \in \mathit{Field}$. Next, we specify constraint rules for the different kinds of statements that may involve records and pointers:

X = { X_1:X'_1, ... , X_k:X'_k }:	$[\![X'_1]\!] \subseteq [\![X\texttt{.}X_1]\!] \;\wedge\; \ldots \;\wedge$ $[\![X'_k]\!] \subseteq [\![X\texttt{.}X_k]\!]$
X = alloc { X_1:X'_1, ... , X_k:X'_k }:	alloc-$i \in [\![X]\!] \;\wedge$ $[\![X'_1]\!] \subseteq [\![\texttt{alloc-}i\texttt{.}X_1]\!] \;\wedge\; \ldots \;\wedge$ $[\![X'_k]\!] \subseteq [\![\texttt{alloc-}i\texttt{.}X_k]\!]$
X_1 = X_2.X_3:	$[\![X_2\texttt{.}X_3]\!] \subseteq [\![X_1]\!]$
X_1 = X_2:	$[\![X_2]\!] \subseteq [\![X_1]\!] \;\wedge$ $[\![X_2\texttt{.}f]\!] \subseteq [\![X_1\texttt{.}f]\!]$ for each $f \in \mathit{Field}$
X_1 = *X_2:	$c \in [\![X_2]\!] \implies ([\![c]\!] \subseteq [\![X_1]\!] \;\wedge\; [\![c\texttt{.}f]\!] \subseteq [\![X_1\texttt{.}f]\!])$ for each $c \in \mathit{Cell}$ and $f \in \mathit{Field}$
X_1.X_2 = X_3: (*X_1).X_2 = X_3:	see Exercise 11.7

As usual, other syntactic forms can be normalized to these basic constructs, and each constraint models the flow of pointer values.

As an example, for the (already normalized) program

```
i = null;
x = alloc {f: i}; // alloc-1
y = alloc {h: x}; // alloc-2
t = *y;
z = t.h;
```

where $\mathit{Cell} = \{\texttt{x}, \texttt{y}, \texttt{z}, \texttt{i}, \texttt{t}, \texttt{alloc-1}, \texttt{alloc-2}\}$ and $\mathit{Field} = \{\texttt{f}, \texttt{h}\}$ we obtain the constraints

$\texttt{alloc-1} \in [\![\texttt{x}]\!]$
$[\![\texttt{i}]\!] \subseteq [\![\texttt{alloc-1.f}]\!]$
$\texttt{alloc-2} \in [\![\texttt{y}]\!]$
$[\![\texttt{x}]\!] \subseteq [\![\texttt{alloc-2.h}]\!]$
$c \in [\![\texttt{y}]\!] \implies ([\![c]\!] \subseteq [\![\texttt{t}]\!] \;\wedge\; [\![c\texttt{.}f]\!] \subseteq [\![\texttt{t.}f]\!])$ for each $c \in \mathit{Cell}$ and $f \in \mathit{Field}$
$[\![\texttt{t.h}]\!] \subseteq [\![z]\!]$

and after computing the least solution we get these points-to sets for the program variables:

$pt(\texttt{x}) = pt(\texttt{z}) = \{\texttt{alloc-1}\}$
$pt(\texttt{y}) = \{\texttt{alloc-2}\}$
$pt(\texttt{i}) = pt(\texttt{t}) = \emptyset$

Exercise 11.7: Specify a suitable constraint rule for both forms of field write statements, $X_1\texttt{.}X_2\texttt{=}X_3\texttt{;}$ and $\texttt{(*}X_1\texttt{).}X_2\texttt{ = }X_3\texttt{;}$. Then explain how the analysis works on this program:

```
x = alloc {f: 1, g: 2};
y = alloc {h: x, g: 3};
z = alloc 4;
(*y).h = z;
```

In Java-like languages, the fields of an object are always accessed via a reference (i.e., a pointer) to the object. The Java syntax $E\texttt{.}X$ for accessing a field X of an object pointed to by an expression E corresponds to the TIP (or C) syntax $\texttt{(*}E\texttt{).}X$.

Exercise 11.8: Assume we restrict the syntax of TIP pointer operations to these Java-like expressions:

$$\begin{array}{lll} \mathit{Exp} & \to & \mathit{Id} \\ & | & \texttt{alloc \{ }\mathit{Id}\texttt{:}\mathit{Exp}\texttt{,} \ldots \texttt{,}\ \mathit{Id}\texttt{:}\mathit{Exp}\texttt{ \}} \\ & | & \texttt{(*}\mathit{Exp}\texttt{).}\mathit{Id} \\ & | & \texttt{null} \end{array}$$

and these statements:

$$\begin{array}{lll} \mathit{Stm} & \to & \mathit{Id}\texttt{ = }\mathit{Exp}\texttt{;} \\ & | & \texttt{(*}\mathit{Exp}\texttt{).}\mathit{Id}\texttt{ = }\mathit{Exp}\texttt{;} \end{array}$$

That is, we can no longer create pointers to variables or to cells containing non-record values but only to heap-allocated records.

Specify constraint rules for Andersen-style points-to analysis for these operations (in a properly normalized form).

11.3 Steensgaard's Algorithm

An interesting alternative is Steensgaard's algorithm [Ste96], which performs a coarser analysis essentially by viewing assignments as being bidirectional. The analysis can be expressed elegantly using term unification; for that reason it is called a *unification-based* points-to analysis. We use a term variable $[\![c]\!]$ for every abstract cell c and a term constructor $\uparrow t$ representing a pointer to t. (Notice the change in notation compared to Section 11.2: here, $[\![c]\!]$ is a term variable and does not directly denote a set of abstract cells.)

X = `alloc` P:	$[\![X]\!] = \uparrow [\![\texttt{alloc-}i]\!]$
X_1 = &X_2:	$[\![X_1]\!] = \uparrow [\![X_2]\!]$
X_1 = X_2:	$[\![X_1]\!] = [\![X_2]\!]$
X_1 = *X_2:	$[\![X_2]\!] = \uparrow \alpha \ \wedge \ [\![X_1]\!] = \alpha$ where α is a fresh term variable
*X_1 = X_2:	$[\![X_1]\!] = \uparrow \alpha \ \wedge \ [\![X_2]\!] = \alpha$ where α is a fresh term variable

Notice that at most two unification constraints are constructed for each pointer operation. (This is different from Andersen's analysis where the last two kinds of statements require a constraint for each abstract cell.)

As usual, term constructors satisfy the general term equality axiom:

$$\uparrow \alpha_1 = \uparrow \alpha_2 \implies \alpha_1 = \alpha_2$$

The resulting points-to function is defined as:

$$\texttt{pt}(p) = \{t \in \mathit{Cell} \mid [\![p]\!] = \uparrow [\![t]\!]\}$$

For the example program from Section 11.2, Steensgaard's algorithm generates the following constraints:

$[\![\texttt{p}]\!] = \uparrow [\![\texttt{alloc-1}]\!]$
$[\![\texttt{x}]\!] = [\![\texttt{y}]\!]$
$[\![\texttt{x}]\!] = [\![\texttt{z}]\!]$
$[\![\texttt{p}]\!] = \uparrow \alpha_1$
$[\![\texttt{z}]\!] = \alpha_1$
$[\![\texttt{p}]\!] = [\![\texttt{q}]\!]$
$[\![\texttt{q}]\!] = \uparrow [\![\texttt{y}]\!]$
$[\![\texttt{x}]\!] = \alpha_2$
$[\![\texttt{p}]\!] = \uparrow \alpha_2$
$[\![\texttt{p}]\!] = \uparrow [\![\texttt{z}]\!]$

We can use the unification algorithm from Section 3.3 to find the resulting points-to sets:

$$pt(\texttt{p}) = pt(\texttt{q}) = \{\texttt{alloc-1}, \texttt{y}, \texttt{z}\}$$

This result is less precise than what we obtained using Andersen's algorithm, but is reached using the faster algorithm.

Notice that unification can never fail in Steensgaard's analysis, since there is only one kind of term constructor (unlike the type analysis in Chapter 3).

Exercise 11.9: Use Steensgaard's algorithm to compute the points-to sets for the two programs from Exercise 11.2 and Exercise 11.3. Are the results less precise than when using Andersen's analysis?

Exercise 11.10: It is tempting to simplify the constraint rule for X_1 `=` `*`X_2`;` from

$$[\![X_2]\!] = \uparrow\alpha \ \wedge \ [\![X_1]\!] = \alpha$$

to

$$[\![X_2]\!] = \uparrow[\![X_1]\!]$$

however, this may degrade the analysis precision. Give an example of a program where this simplification affects the analysis results.

Similarly, can the constraint rule for `*`X_1 `=` X_2`;` be simplified from

$$[\![X_1]\!] = \uparrow\alpha \ \wedge \ [\![X_2]\!] = \alpha$$

to

$$[\![X_1]\!] = \uparrow[\![X_2]\!]$$

without affecting the analysis results?

Exercise 11.11: Prove that Andersen's analysis is always at least as precise as Steensgaard's analysis.

11.4 Interprocedural Pointer Analysis

In languages with both function values and pointers, functions 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 value accessed via a 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_0`(`X_1`,`$\ldots$`,` X_n`);`

so that the involved expressions are all variables. Similarly, all return expressions are assumed to be just variables, as in `return` X`;`.

Exercise 11.12: 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

$$f \in [\![X_0]\!] \implies ([\![X_1]\!] \subseteq [\![X_1']\!] \wedge \cdots \wedge [\![X_n]\!] \subseteq [\![X_n']\!] \wedge [\![X']\!] \subseteq [\![X]\!])$$

for every occurrence of a function definition with n parameters

f`(`X_1'`,...,`X_n'`)` { ...`return` X'`;` }

This will maintain the precision of the control flow analysis.

Exercise 11.13: Design a *context sensitive* variant of the Andersen-style points-to analysis. (Hint: see Sections 8.2–8.4.)

Exercise 11.14: Continuing Exercise 11.13, can we still use the cubic algorithm (Section 10.2) 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?

11.5 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 11.2. The basic lattice we use, called $Null$, is:

where the bottom element `NN` means *definitely not* `null` and the top element $\top$ represents values that may be `null`. We then form the following map lattice for abstract states:

$$State = Cell \to Null$$

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:

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

where

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

For a 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 load operations:

$$X_1 \texttt{ = *}X_2\texttt{:} \quad [\![v]\!] = load(JOIN(v), X_1, X_2)$$

where

$$load(\sigma, X_1, X_2) = \sigma[X_1 \mapsto \bigsqcup_{\alpha \in pt(X_2)} \sigma(\alpha)]$$

Similar reasoning gives constraints for the other operations that affect pointer variables:

$$\begin{array}{rll} X \texttt{ = alloc } P\texttt{:} & [\![v]\!] & = JOIN(v)[X \mapsto \texttt{NN}, \texttt{alloc-}i \mapsto \top] \\ X_1 \texttt{ = \&}X_2\texttt{:} & [\![v]\!] & = JOIN(v)[X_1 \mapsto \texttt{NN}] \\ X_1 \texttt{ = } X_2\texttt{:} & [\![v]\!] & = JOIN(v)[X_1 \mapsto JOIN(v)(X_2)] \\ X \texttt{ = null:} & [\![v]\!] & = JOIN(v)[X \mapsto \top] \end{array}$$

Exercise 11.15: Explain why the above four constraints are monotone and sound.

For a 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 store operations, we must therefore join the new abstract value into the existing one for each affected cell in $pt(X_1)$:

$$\texttt{*}X_1 \texttt{ = } X_2\texttt{:} \quad [\![v]\!] = store(JOIN(v), X_1, X_2)$$

where

$$store(\sigma, X_1, X_2) = \sigma\ [\alpha \mapsto \sigma(\alpha) \sqcup \sigma(X_2)\ ]_{\alpha \in pt(X_1)}$$

The situation we here see at 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 a 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 null;
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:

$[\![\texttt{p = alloc null}]\!] = \bot[\texttt{p} \mapsto \texttt{NN}, \texttt{alloc-1} \mapsto \top]$
$[\![\texttt{q = \&p}]\!] = [\![\texttt{p = alloc null}]\!][\texttt{q} \mapsto \texttt{NN}]$
$[\![\texttt{n = null}]\!] = [\![\texttt{q = \&p}]\!][\texttt{n} \mapsto \top]$
$[\![\texttt{*q = n}]\!] = [\![\texttt{n = null}]\!][\texttt{p} \mapsto [\![\texttt{n = null}]\!](\texttt{p}) \sqcup [\![\texttt{n = null}]\!](\texttt{n})]$
$[\![\texttt{*p = n}]\!] = [\![\texttt{*q = n}]\!][\texttt{alloc-1} \mapsto [\![\texttt{*q = n}]\!](\texttt{alloc-1}) \sqcup [\![\texttt{*q = n}]\!](\texttt{n})]$

The least solution is:

$[\![\texttt{p = alloc null}]\!] = [\texttt{p} \mapsto \texttt{NN}, \texttt{q} \mapsto \texttt{NN}, \texttt{n} \mapsto \texttt{NN}, \texttt{alloc-1} \mapsto \top]$
$[\![\texttt{q = \&p}]\!] = [\texttt{p} \mapsto \texttt{NN}, \texttt{q} \mapsto \texttt{NN}, \texttt{n} \mapsto \texttt{NN}, \texttt{alloc-1} \mapsto \top]$
$[\![\texttt{n = null}]\!] = [\texttt{p} \mapsto \texttt{NN}, \texttt{q} \mapsto \texttt{NN}, \texttt{n} \mapsto \top, \texttt{alloc-1} \mapsto \top]$
$[\![\texttt{*q = n}]\!] = [\texttt{p} \mapsto \top, \texttt{q} \mapsto \texttt{NN}, \texttt{n} \mapsto \top, \texttt{alloc-1} \mapsto \top]$
$[\![\texttt{*p = n}]\!] = [\texttt{p} \mapsto \top, \texttt{q} \mapsto \texttt{NN}, \texttt{n} \mapsto \top, \texttt{alloc-1} \mapsto \top]$

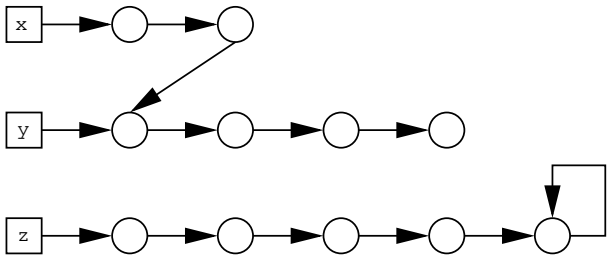
By inspecting this information, an analysis could statically detect that when `*q = n` is evaluated, which is immediately after `n = null`, the variable `q` is definitely non-`null`. On the other hand, when `*p = n` is evaluated, we cannot rule out the possibility that `p` may contain `null`.

Exercise 11.16: Show an alternative constraint for 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 11.17: Show an (unsound) alternative constraint for store operations using strong update, together with an example program where the modified analysis then gives a wrong result.

11.6 Flow-Sensitive Pointer Analysis

Note that we can produce interesting heap structures with TIP programs, even without records. 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 an 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

$$State = \mathcal{P}(Cell \times Cell)$$

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:

$$\begin{array}{rll}
X \texttt{ = alloc } P\texttt{:} & [\![v]\!] & = JOIN(v)\!\downarrow\! X \cup \{(X, \texttt{alloc-}i)\} \\
X_1 \texttt{ = \&}X_2\texttt{:} & [\![v]\!] & = JOIN(v)\!\downarrow\! X_1 \cup \{(X_1, X_2)\} \\
X_1 \texttt{ = } X_2\texttt{:} & [\![v]\!] & = assign(JOIN(v), X_1, X_2) \\
X_1 \texttt{ = *}X_2\texttt{:} & [\![v]\!] & = load(JOIN(v), X_1, X_2) \\
\texttt{*}X_1 \texttt{ = } X_2\texttt{:} & [\![v]\!] & = store(JOIN(v), X_1, X_2) \\
X \texttt{ = null:} & [\![v]\!] & = JOIN(v)\!\downarrow\! X
\end{array}$$

and for all other nodes:

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

where

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

$$\sigma{\downarrow}x = \{(s,t) \in \sigma \mid s \neq x\}$$

$$assign(\sigma, x, y) = \sigma{\downarrow}x \ \cup \{(x,t) \mid (y,t) \in \sigma\}$$

$$load(\sigma, x, y) = \sigma{\downarrow}x \ \cup \{(x,t) \mid (y,s) \in \sigma, (s,t) \in \sigma\}$$

$$store(\sigma, x, y) = \sigma \ \cup \{(s,t) \mid (x,s) \in \sigma, (y,t) \in \sigma\}$$

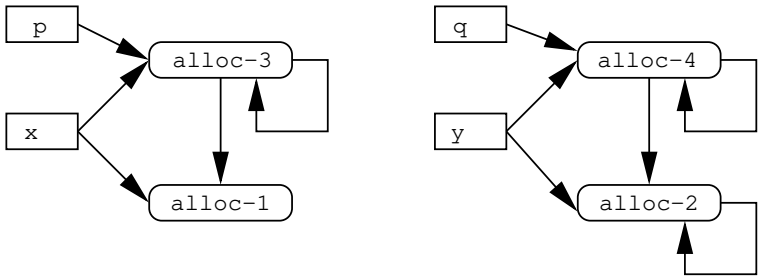
Notice that the constraint for store operations uses weak update.

Exercise 11.18: Explain the above constraints.

Consider now the following program:

```
var x,y,n,p,q;
x = alloc null; y = alloc null;
*x = null; *y = y;
n = input;
while (n>0) {
  p = alloc null; q = alloc null;
  *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:

$$pt_v(p) = \{t \mid (p,t) \in [\![v]\!]\}$$

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;
t = x;
```

Andersen's algorithm would predict that $pt(\mathtt{t}) = \{\mathtt{y}, \mathtt{z}\}$ whereas the flow-sensitive analysis computes $pt_v(\mathtt{t}) = \{\mathtt{z}\}$ for the final program point v.

11.7 Escape Analysis

In Section 3.5 we 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 cells that return expressions may point to in the points-to graph cannot reach arguments or variables defined in the function itself, since all other locations must then necessarily be heap-allocated or reside in earlier frames on the invocation stack.

Chapter 12

Abstract Interpretation

In the preceding chapters we have used the term *soundness* of an analysis only informally: if an analysis is sound, the properties it infers for a given program hold in all actual executions of the program. The theory of abstract interpretation provides a solid mathematical foundation for what it means for an analysis to be sound, by relating the analysis specification to the formal semantics of the programming language. Another use of abstract interpretation is for understanding whether an analysis design, or a part of an analysis design, is as precise as possible relative to a choice of analysis lattice and where imprecision may arise. The fundamental ideas of abstract interpretation were introduced by Cousot and Cousot in the 1970s [CC76, CC77, CC79b].

12.1 A Collecting Semantics for TIP

We begin by defining formal semantics of the same subset of TIP that we used for the sign analysis in Sections 4.1 and 5.1, meaning that we ignore function calls, pointers, and records. In this process we clarify some of the under-specified parts of TIP as discussed in Exercise 2.1. Instead of using traditional styles of semantics, such as operational semantics, denotational semantics, or axiomatic semantics, we choose a constraint-based approach that aligns well with our formulations of the analyses presented in the preceding chapters. Moreover, we choose to define the semantics based on the CFG representation of TIP programs. These choices allow us to more concisely relate the semantics and the analysis. What matters is that the semantics captures the meaning of programs in ordinary executions, without any approximations. The semantics specifies how a *concrete* program execution works, whereas our analyses can be thought of as *abstract* interpreters.[1]

[1] The research literature on abstract interpretation sometimes refers to what we here call semantics as the "concrete semantics" and the analysis specification is called the "abstract semantics".

A concrete state is a partial map from program variables to integers:[2]

$$\mathit{ConcreteState} = \mathit{Var} \hookrightarrow \mathbb{Z}$$

For every CFG node v we have a constraint variable that ranges over sets of concrete states:

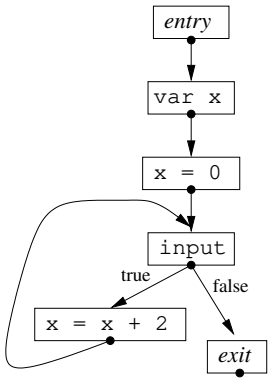
$$\{\!|v|\!\} \subseteq \mathit{ConcreteState}$$

The idea is that $\{\!|v|\!\}$ shall denote the set of concrete states that are possible at the program point immediately after the instruction represented by v, in some execution of the program. This is called a *collecting semantics*, because it "collects" the possible states. In Exercises 12.59–12.64 we shall study other kinds of collecting semantics that collect relevant information suitable for other analyses, such as live variables analysis. We choose to focus on the program point immediately *after* the instruction of the CFG node, instead of the program point before, to align with our sign analysis and the other forward analyses from Chapter 5.

Consider this simple program as an example:

```
var x;
x = 0;
while (input) {
  x = x + 2;
}
```

Its CFG looks as follows, where the bullets represent the program points that have associated constraint variables.



The solution we are interested in maps the constraint variable $\{\!|\texttt{x = 0}|\!\}$ to the single state where `x` is zero and $\{\!|\texttt{x = x + 2}|\!\}$ to the set of all states where `x` is a positive even number, and similarly for the other program points.

As a first step, we define some useful auxiliary functions, ceval, csucc, and CJOIN, that have a close connection to the auxiliary functions used in the

[2]We use the notation $A \hookrightarrow B$ to denote the set of partial functions from A to B.

specification of the sign analysis, but now for concrete executions instead of abstract executions.

The function $ceval\colon \mathit{ConcreteState} \times \mathit{Exp} \to \mathcal{P}(\mathbb{Z})$ gives the semantics of evaluating an expression $E \in \mathit{Exp}$ relative to a concrete state $\rho \in \mathit{ConcreteState}$, which results in a set of possible integer values, defined inductively as follows:[3]

$$
\begin{array}{l}
ceval(\rho, X) = \begin{cases} \{\rho(X)\} & \text{if } X \in \mathrm{dom}(\rho) \\ \emptyset & \text{otherwise} \end{cases} \\
ceval(\rho, I) = \{I\} \\
ceval(\rho, \texttt{input}) = \mathbb{Z} \\
ceval(\rho, E_1 \texttt{ + } E_2) = \{z_1 + z_2 \mid z_1 \in ceval(\rho, E_1) \ \wedge \ z_2 \in ceval(\rho, E_2)\} \\
ceval(\rho, E_1 \texttt{ / } E_2) = \{z_1 \ / \ z_2 \mid z_1 \in ceval(\rho, E_1) \ \wedge \ z_2 \in ceval(\rho, E_2)\}
\end{array}
$$

Evaluation of the other binary operators is defined similarly. In this simple subset of TIP we consider here, evaluating an expression cannot affect the values of the program variables. Also note that division by zero simply results in the empty set of values.

We overload $ceval$ such that it also works on *sets* of concrete states, $ceval\colon \mathcal{P}(\mathit{ConcreteState}) \times \mathit{Exp} \to \mathcal{P}(\mathbb{Z})$:

$$ceval(R, E) = \bigcup_{\rho \in R} ceval(\rho, E)$$

The function $csucc\colon \mathit{ConcreteState} \times \mathit{Node} \to \mathcal{P}(\mathit{Node})$ gives the set of possible successors of a CFG node relative to a concrete state. If v is an `if` or `while` node with branch condition E and $\rho \in \mathit{ConcreteState}$, then $csucc(\rho, v)$ contains v's *true* successor in the CFG if $z \in ceval(\rho, E)$ for some $z \neq 0$, it contains v's *false* successor if $0 \in ceval(\rho, E)$, and it contains no other nodes. (Since evaluation of expressions does not affect the values of the program variables, as noted above, conveniently it does not matter whether the states given as the first argument to $csucc$ belong to the program point immediately before or immediately after the `if`/`while` node.) Note that $csucc$ may return two successors, even for a single concrete state (if the branch condition contains `input`), and it also may return zero successors (in case of division by zero). For all other kinds of nodes, let $csucc(\rho, v) = succ(v)$. Similar to the definition of $ceval$, we also overload $csucc$ to work on sets of concrete states, $csucc\colon \mathcal{P}(\mathit{ConcreteState}) \times \mathit{Node} \to \mathcal{P}(\mathit{Node})$:

$$csucc(R, v) = \bigcup_{\rho \in R} csucc(\rho, v)$$

For a CFG node v, $CJOIN(v)$ denotes the set of states at the program point immediately *before* the instruction represented by v, relative to the states at the program points *after* the relevant other nodes according to the $csucc$ function:[4]

$$CJOIN(v) = \{\rho \in \mathit{ConcreteState} \mid \exists w \in \mathit{Node}\colon \rho \in \{\!\![w]\!\!\} \wedge v \in csucc(\rho, w)\}$$

[3] We let I denote both an arbitrary syntactic numeral and the mathematical integer it describes, and for simplicity we do not restrict the numeric computations to, for example, 64-bit signed integers.

[4] Note that $CJOIN(v)$ is a function of all the constraint variables $\{\!\![v_1]\!\!\}, \ldots, \{\!\![v_n]\!\!\}$ for the entire program, just like $JOIN(v)$ is a function of all the constraint variables $[\![v_1]\!], \ldots, [\![v_n]\!]$.

Exercise 12.1: Convince yourself that this definition of $CJOIN$ makes sense, especially for the cases where v is a node with multiple incoming edges (like `input` in the example on page 164) or it is the first node of a branch (like `x = x + 2` in the example).

The semantics of a node v that represents an assignment statement *X* = *E* can now be expressed as the following constraint rule:

$$[\![X{=}E]\!] = \left\{\rho[X \mapsto z] \;\middle|\; \rho \in CJOIN(v) \;\wedge\; z \in ceval(\rho, E)\right\}$$

This rule formalizes the runtime behavior of assignments: For every state ρ that may appear immediately before executing the assignment, the state after the assignment is obtained by overwriting the value of X with the result of evaluating E.

If v is a variable declaration, `var` X_1`,` $\ldots$ `,`X_n, we use this rule:

$$[\![\texttt{var}\ X_1, \ldots, X_n]\!] = \left\{\rho[X_1 \mapsto z_1, \ldots, X_n \mapsto z_n] \;\middle|\; \rho \in CJOIN(v) \wedge z_1 \in \mathbb{Z} \wedge \cdots \wedge z_n \in \mathbb{Z}\right\}$$

Ignoring parameters of `main`, the only possible initial state at entry nodes is the partial map that is undefined for all program variables, denoted $[]$:

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

For all other kinds of nodes, we have this trivial constraint rule:

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

Notice the resemblance with the analysis constraints from Section 5.1.

Exercise 12.2: Define a suitable constraint rule that expresses the semantics of `assert` statements (see Section 7.1) in our collecting semantics. (For this exercise, think of `assert` statements as written explicitly by the programmers anywhere in the programs, not just for use in control sensitive analysis.)

Conveniently, the set of valuations of each constraint variable forms a powerset lattice (see Section 4.3), $\mathcal{P}(ConcreteState)$ ordered by subset. For the entire program, we thus have the product lattice $\left(\mathcal{P}(ConcreteState)\right)^n$ where n is the number of CFG nodes for the program. The powerset of concrete values, $\mathcal{P}(\mathbb{Z})$, similarly forms a complete lattice.

A program with n CFG nodes, $v_1, \ldots, v_n$, is thus represented by n equations:

$$\begin{aligned}
[\![v_1]\!] &= cf_{v_1}([\![v_1]\!], \ldots, [\![v_n]\!]) \\
[\![v_2]\!] &= cf_{v_2}([\![v_1]\!], \ldots, [\![v_n]\!]) \\
&\vdots \\
[\![v_n]\!] &= cf_{v_n}([\![v_1]\!], \ldots, [\![v_n]\!])
\end{aligned}$$

The constraint for each CFG node v is a function $cf_v \colon \big(\mathcal{P}(ConcreteState)\big)^n \to \mathcal{P}(ConcreteState)$, much like the analysis of a program is expressed as an equation system in Sections 4.4 and 5.1. In the same way, we can combine the n functions into one, $cf \colon \big(\mathcal{P}(ConcreteState)\big)^n \to \big(\mathcal{P}(ConcreteState)\big)^n$, defined by

$$cf(x_1, \ldots, x_n) = \big(cf_{v_1}(x_1, \ldots, x_n), \ldots, cf_{v_n}(x_1, \ldots, x_n)\big)$$

in which case the equation system looks like

$$x = cf(x)$$

where $x \in \big(\mathcal{P}(ConcreteState)\big)^n$. Thus, cf captures the semantics of the program as one single function, in the same way af in Chapter 5 (page 54) captures the analysis of the program as one single function.

Exercise 12.3: Show that the constraints defined by the rules above are monotone. Then show that the combined constraint function cf is also monotone.

With these definitions and observations, we can define the semantics of a given program as the least solution to the generated constraints. A solution to a constraint system is, as usual, a valuation of the constraint variables that satisfies all the constraints – in other words, a fixed point of cf. Since the function cf is monotone according to Exercise 12.3, Tarski's fixed-point theorem from Chapter 6 (page 81) immediately gives us that it has a unique least fixed point, thus a unique least solution to the constraints always exists.

To motivate why we are interested in the *least* solution, consider again the example program from page 164. It only contains one variable, so $Var = \{\texttt{x}\}$. Here are two different solutions to the semantic constraints:

	solution 1	solution 2
$\{\!\![\textit{entry}]\!\!\}$	$\{[]\}$	$\{[]\}$
$\{\!\![\texttt{var x}]\!\!\}$	$\{[\texttt{x} \mapsto z] \mid z \in \mathbb{Z}\}$	$\{[\texttt{x} \mapsto z] \mid z \in \mathbb{Z}\}$
$\{\!\![\texttt{x = 0}]\!\!\}$	$\{[\texttt{x} \mapsto 0]\}$	$\{[\texttt{x} \mapsto 0]\}$
$\{\!\![\texttt{input}]\!\!\}$	$\{[\texttt{x} \mapsto z] \mid z \in \{0, 2, 4, \ldots\}\}$	$\{[\texttt{x} \mapsto z] \mid z \in \mathbb{Z}\}$
$\{\!\![\texttt{x = x + 2}]\!\!\}$	$\{[\texttt{x} \mapsto z] \mid z \in \{2, 4, \ldots\}\}$	$\{[\texttt{x} \mapsto z] \mid z \in \mathbb{Z}\}$
$\{\!\![\textit{exit}]\!\!\}$	$\{[\texttt{x} \mapsto z] \mid z \in \{0, 2, 4, \ldots\}\}$	$\{[\texttt{x} \mapsto z] \mid z \in \mathbb{Z}\}$

Naturally, for this particular program we want a solution where `x` in the loop and at the exit point can only be a nonnegative even integer, not an arbitrary integer.

Exercise 12.4:

(a) Check that both of the above solutions are indeed solutions to the constraints for this particular program.

(b) Give an example of yet another solution.

(c) Argue that solution 1 is the least of all solutions.

Recall the example program from Sections 4.1 and 5.1:

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

For this program, at the program points immediately after the assignment `b = 87`, immediately after the assignment `c = a - b` (at the end of the `else` branch), and the exit, the following sets of concrete states are possible according to the collecting semantics:

$\{\![\texttt{b = 87}]\!\} = \{[\texttt{a} \mapsto 42, \texttt{b} \mapsto 87, \texttt{c} \mapsto z] \mid z \in \mathbb{Z}\}$
$\{\![\texttt{c = a - b}]\!\} = \{[\texttt{a} \mapsto 42, \texttt{b} \mapsto 87, \texttt{c} \mapsto -45]\}$
$\{\![\mathit{exit}]\!\} = \{[\texttt{a} \mapsto 42, \texttt{b} \mapsto 87, \texttt{c} \mapsto 129], [\texttt{a} \mapsto 42, \texttt{b} \mapsto 87, \texttt{c} \mapsto -45]\}$

Exercise 12.5: Check that the least fixed point of the semantic constraints for the program is indeed these sets, for the three program points.

In comparison, the sign analysis specified in Section 5.1 computes the following abstract states for the same program points:

$[\![\texttt{b = 87}]\!] = [\texttt{a} \mapsto +, \texttt{b} \mapsto +, \texttt{c} \mapsto \top]$
$[\![\texttt{c = a - b}]\!] = [\texttt{a} \mapsto +, \texttt{b} \mapsto +, \texttt{c} \mapsto \top]$
$[\![\mathit{exit}]\!] = [\texttt{a} \mapsto +, \texttt{b} \mapsto +, \texttt{c} \mapsto \top]$

In this specific case the analysis result is almost the best we could hope for, with that choice of analysis lattice. Notice that the abstract value of `c` at the program point after `c = a - b` is $\top$, although the only possible value in concrete executions is -45. This is an example of a conservative analysis result.

As shown above, Tarski's fixed-point theorem ensures that the collecting semantics is well defined – even though it is generally non-computable. The variant of Kleene's fixed-point theorem from Chapter 4 (page 47) is not strong enough to ensure this property, as the lattice has infinite height.

Exercise 12.6: Show that the lattice $(\mathcal{P}(\mathit{ConcreteState}))^n$ has infinite height.

An alternative to using Tarski's fixed-point theorem is to use the fact that Kleene's fixed-point theorem also holds for infinite-height lattices when f preserves arbitrary nonempty joins, that is, $f(\bigsqcup A) = \bigsqcup_{a \in A} f(a)$ for every nonempty $A \subseteq L$. Note that if f preserves arbitrary nonempty joins, it is also monotone.[5] All the constraints for the collecting semantics defined by the rules above are not just monotone but preserve arbitrary nonempty joins.

[5] For a variant of Kleene's theorem that only requires monotonicity, see Exercise 4.32.

Exercise 12.7: Prove that functions that preserve arbitrary nonempty joins are monotone.

Exercise 12.8: Prove that if L is a complete lattice (not necessarily with finite height) and the function $f : L \to L$ preserves arbitrary nonempty joins, then $\mathit{lfp}(f) = \bigsqcup_{i \geq 0} f^i(\bot)$ is the least fixed point for f.

Exercise 12.9: Show that the constraints defined by the rules above for the collecting semantics, including the combined constraint function cf, indeed preserve arbitrary nonempty joins.

Exercise 12.10: For readers familiar with traditional operational semantics, denotational semantics, or axiomatic semantics: Specify the semantics for the same subset of TIP as considered above, but this time using operational semantics, denotational semantics, or axiomatic semantics. Your semantics and the one specified above should be equivalent in the following sense: For every program P, at every program point p in P, the set of concrete states that may appear at p in some execution of P is the same for the two different styles of semantics. Then prove that your semantics has this property. (We continue with this topic in Section 12.6.)

For use later in this chapter, let us introduce the notation $\{\!|P|\!\} = \mathit{lfp}(cf)$ and $[\![P]\!] = \mathit{lfp}(af)$ where cf is the semantic constraint function and af is the analysis constraint function for a given program P. In other words, $\{\!|P|\!\}$ denotes the semantics of P, and $[\![P]\!]$ denotes the analysis result for P.

12.2 Abstraction and Concretization

To clarify the connection between *concrete* information and *abstract* information for the sign analysis example, let us consider three different *abstraction functions* that tell us how each element from the semantic lattices is most precisely described by an element in the analysis lattices. The functions map sets of concrete values, sets of concrete states, and n-tuples of sets of concrete states to their most precise possible abstract counterparts:

$$\alpha_{\mathsf{a}} : \mathcal{P}(\mathbb{Z}) \to \mathit{Sign}$$
$$\alpha_{\mathsf{b}} : \mathcal{P}(\mathit{ConcreteState}) \to \mathit{State}$$
$$\alpha_{\mathsf{c}} : \big(\mathcal{P}(\mathit{ConcreteState})\big)^n \to \mathit{State}^n$$

As before, $\mathcal{P}(\mathbb{Z})$ is the powerset lattice defined over the set of integers ordered by subset, Sign is the sign lattice from Section 4.1, we define $\mathit{ConcreteState} = \mathit{Var} \hookrightarrow \mathbb{Z}$ and $\mathit{State} = \mathit{Var} \to \mathit{Sign}$ as in Sections 12.1 and 4.1, respectively, and

n is the number of CFG nodes. The functions are defined as follows, to precisely capture the informal descriptions given earlier:

$$\alpha_{\mathsf{a}}(D) = \begin{cases} \bot & \text{if } D \text{ is empty} \\ \texttt{+} & \text{if } D \text{ is nonempty and contains only positive integers} \\ \texttt{-} & \text{if } D \text{ is nonempty and contains only negative integers} \\ \texttt{0} & \text{if } D \text{ is nonempty and contains only the integer } 0 \\ \top & \text{otherwise} \end{cases}$$
for any $D \in \mathcal{P}(\mathbb{Z})$

$\alpha_{\mathsf{b}}(R) = \sigma$ where $\sigma(X) = \alpha_{\mathsf{a}}(\{\rho(X) \mid \rho \in R, X \in \mathrm{dom}(\rho)\})$
for any $R \subseteq \mathit{ConcreteState}$ and $X \in \mathit{Var}$

$\alpha_{\mathsf{c}}(R_1, \ldots, R_n) = (\alpha_b(R_1), \ldots, \alpha_b(R_n))$
for any $R_1, \ldots, R_n \subseteq \mathit{ConcreteState}$

It is a natural condition that abstraction functions are monotone. Intuitively, a larger set of concrete values or states should not be represented by a smaller abstract element in the lattice order.

Exercise 12.11: Argue that the three functions α_{a}, α_{b}, and α_{c} defined above for the sign analysis are monotone.

Dually, we may define *concretization functions* that express the meaning of the analysis lattice elements in terms of the concrete values, states, and n-tuples of states:

$\gamma_{\mathsf{a}} \colon \mathit{Sign} \to \mathcal{P}(\mathbb{Z})$
$\gamma_{\mathsf{b}} \colon \mathit{State} \to \mathcal{P}(\mathit{ConcreteState})$
$\gamma_{\mathsf{c}} \colon \mathit{State}^n \to \big(\mathcal{P}(\mathit{ConcreteState})\big)^n$

defined by

$$\gamma_{\mathsf{a}}(s) = \begin{cases} \emptyset & \text{if } s = \bot \\ \{1, 2, 3, \ldots\} & \text{if } s = \texttt{+} \\ \{-1, -2, -3, \ldots\} & \text{if } s = \texttt{-} \\ \{0\} & \text{if } s = \texttt{0} \\ \mathbb{Z} & \text{if } s = \top \end{cases}$$
for any $s \in \mathit{Sign}$

$\gamma_{\mathsf{b}}(\sigma) = \{\rho \in \mathit{ConcreteState} \mid \rho(X) \in \gamma_{\mathsf{a}}(\sigma(X)) \text{ for all } X \in \mathrm{dom}(\rho)\}$
for any $\sigma \in \mathit{State}$

$\gamma_{\mathsf{c}}(\sigma_1, \ldots, \sigma_n) = (\gamma_{\mathsf{b}}(\sigma_1), \ldots, \gamma_{\mathsf{b}}(\sigma_n))$
for any $(\sigma_1, \ldots, \sigma_n) \in \mathit{State}^n$

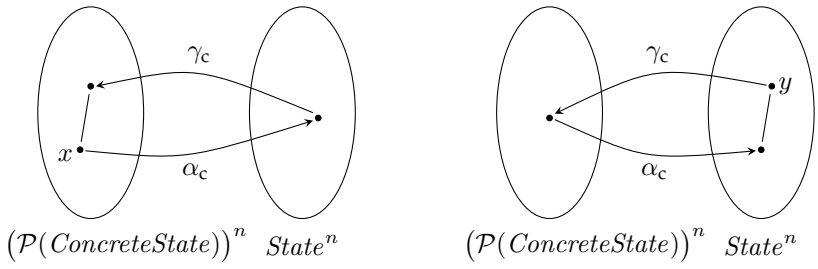
Concretization functions are, like abstraction functions, naturally monotone.

Exercise 12.12: Argue that the three functions γ_{a}, γ_{b}, and γ_{c} from the sign analysis example are monotone.

Furthermore, abstraction functions and concretization functions that arise naturally when developing program analyses are closely connected. If L_1 and L_2 are complete lattices, $\alpha\colon L_1 \to L_2$ is an abstraction function, and $\gamma\colon L_2 \to L_1$ is a concretization function, then α and γ usually have the following properties:

- $\gamma \circ \alpha$ is extensive (meaning that $x \sqsubseteq \gamma(\alpha(x))$ for all $x \in L_1$; see Exercise 4.18), and
- $\alpha \circ \gamma$ is reductive (meaning that $\alpha(\gamma(y)) \sqsubseteq y$ for all $y \in L_2$).

The pair of monotone functions, α and γ, is called a *Galois connection* if it satisfies these two properties. The intuition of the first property is that abstraction may lose precision but must be safe. One way to interpret the second property is that the abstraction function should always give the most precise possible abstract description for any element in the semantic lattice. In many cases, $\alpha \circ \gamma$ is the identity function. The two properties can be illustrated as follows, using α_{c} and γ_{c} from the sign analysis as an example:



Exercise 12.13: Show that all three pairs of abstraction and concretization functions $(\alpha_{\mathsf{a}}, \gamma_{\mathsf{a}})$, $(\alpha_{\mathsf{b}}, \gamma_{\mathsf{b}})$, and $(\alpha_{\mathsf{c}}, \gamma_{\mathsf{c}})$ from the sign analysis example are Galois connections. (See also Exercise 12.29.)

Exercise 12.14: Show that $\alpha_{\mathsf{a}} \circ \gamma_{\mathsf{a}}$, $\alpha_{\mathsf{b}} \circ \gamma_{\mathsf{b}}$, and $\alpha_{\mathsf{c}} \circ \gamma_{\mathsf{c}}$ are all equal to the identity function, for the three pairs of abstraction and concretization functions from the sign analysis.

Exercise 12.15: Argue that $\gamma \circ \alpha$ is typically *not* the identity function, when $\alpha\colon L_1 \to L_2$ is an abstraction function and $\gamma\colon L_2 \to L_1$ is the associated concretization function for some analysis. (Hint: consider α_{a} and γ_{a} from the sign analysis example.)

Exercise 12.16: Give an example of an analysis with abstraction function α and concretization function γ, such that $\alpha \circ \gamma$ is *not* the identity function.

Exercise 12.17: Assume L_1 and L_2 are complete lattices and α and γ are functions, $\alpha\colon L_1 \to L_2$, and $\gamma\colon L_2 \to L_1$. The pair of functions, α and γ, is called an *adjunction* between L_1 and L_2 if the following condition is satisfied:

$$\forall x \in L_1, y \in L_2\colon \alpha(x) \sqsubseteq y \iff x \sqsubseteq \gamma(y)$$

Prove that α and γ form a Galois connection (i.e., α and γ are monotone, $\gamma \circ \alpha$ is extensive, and $\alpha \circ \gamma$ is reductive) if and only if they are an adjunction between L_1 and L_2.

This theorem is useful for some of the later exercises in this section.

Exercise 12.18: Show that if $\alpha\colon L_1 \to L_2$ and $\gamma\colon L_2 \to L_1$ form a Galois connection between two complete lattices L_1 and L_2, then α is a complete join morphism, i.e. $\alpha(\bigsqcup A) = \bigsqcup_{a \in A} \alpha(a)$ for every $A \subseteq L_1$.

(Hint: see Exercise 12.17.)

We shall use this result in the soundness argument in Section 12.3. Not surprisingly, the dual property also holds: γ satisfies $\gamma(\bigsqcap B) = \bigsqcap_{b \in B} \gamma(b)$ for every $B \subseteq L_2$ when α and γ form a Galois connection. (A function γ satisfying this property is called a *complete meet morphism*.)

Exercise 12.19: Show that if α and γ form a Galois connection, then $\alpha(\bot) = \bot$ and $\gamma(\top) = \top$.

(Hint: see Exercises 4.9 and 12.18.)

We have argued that the Galois connection property is natural for any reasonable pair of an abstraction function and a concretization function, including those that appear in our sign analysis example. The following exercise tells us that it always suffices to specify either α or γ, then the other is uniquely determined if requiring that they together form a Galois connection.

Exercise 12.20: Prove the following properties about Galois connections:

If L_1 and L_2 are complete lattices and the functions $\alpha\colon L_1 \to L_2$ and $\gamma\colon L_2 \to L_1$ form a Galois connection, then γ is uniquely determined by α:

$$\gamma(y) = \bigsqcup\{x \in L_1 \mid \alpha(x) \sqsubseteq y\} \quad \text{for all } y \in L_2$$

Conversely, α is uniquely determined by γ:

$$\alpha(x) = \bigsqcap\{y \in L_2 \mid x \sqsubseteq \gamma(y)\} \quad \text{for all } x \in L_1$$

(Hint: see Exercise 12.17.)

The result from Exercise 12.20 means that once the analysis designer has specified the collecting semantics and the analysis lattice and constraint rules, then the relation between the semantic domain and the analysis domain may be specified using an abstraction function α (resp. a concretization function γ), and then the associated concretization function γ (resp. abstraction function α) is uniquely determined – provided that one exists such that the two functions form a Galois connection.

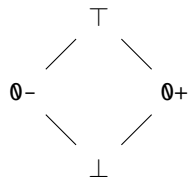
This raises an interesting question: Under what conditions does α (resp. γ) have a corresponding γ (resp. α) such that α and γ form a Galois connection? One answer is that the converse of the property shown in Exercise 12.18 holds too, as shown in the following exercise.

Exercise 12.21: Show that if L_1 and L_2 are complete lattices and $\alpha\colon L_1 \to L_2$ is a complete join morphism, then there exists a function $\gamma\colon L_2 \to L_1$ such that α and γ form a Galois connection.

The dual property also holds: if γ satisfies $\gamma(\bigsqcap B) = \bigsqcap_{b\in B} \gamma(b)$ for every $B \subseteq L_2$ (i.e., γ is a complete meet morphism) then there exists a function $\alpha\colon L_1 \to L_2$ such that α and γ form a Galois connection.

The following exercise demonstrates that the Galois connection property can be used as a "sanity check" when designing analysis lattices.

Exercise 12.22: Instead of using the usual *Sign* lattice from Section 5.1, assume we chose to design our sign analysis based on this lattice:



with the meaning of the elements expressed by this concretization function:

$$\gamma_{\mathrm{a}}(s) = \begin{cases} \emptyset & \text{if } s = \bot \\ \{0, 1, 2, 3, \dots\} & \text{if } s = \texttt{0+} \\ \{0, -1, -2, -3, \dots\} & \text{if } s = \texttt{0-} \\ \mathbb{Z} & \text{if } s = \top \end{cases}$$

At first, this may seem like a reasonable lattice for an analysis, but there is something strange about it: How should we define $eval(\sigma, \texttt{0})$? Or equivalently, how should we define $\alpha_{\mathrm{a}}(\{0\})$? We could somewhat arbitrarily choose either `0+` or `0-`.

Show that, with this choice of lattice, there does not exist an abstraction function α_{a} such that α_{a} and γ_{a} form a Galois connection.

Despite the lack of a Galois connection in the example in Exercise 12.22, in this specific case we could go ahead and design a variant of the sign analysis based on this alternative lattice, without sacrificing the soundness or termination properties. However, many of the useful properties discussed in the following sections might not hold without the Galois connection property, and the analysis might produce potentially surprising results. As an example, assume we analyze the following program using control-insensitive sign analysis (i.e., a sign analysis that ignores branch conditions) with the modified lattice from Exercise 12.22, and that the analysis designer has chosen $\alpha_{\mathrm{a}}(\{0\}) = \texttt{0-}$ and $\alpha_{\mathrm{a}}(\{0, 1\}) = \texttt{0+}$.

```
x = input > 3;
if (!x) {
  output x;
}
```

After the first statement, the abstract value of `x` is `0+` (since > yields 0 or 1), so at the `output` statement, the abstract value of `x` is then `0+`. This means that the analysis is precise enough to be able to conclude that the output value is definitely not a negative number. Now consider a situation where the analysis designer wants to increase analysis precision generally by adding control sensitivity to the analysis, using the approach presented in Chapter 7. With the representation of

branch conditions as `assert` statements, the simple branch condition `!x` can be modeled by the following rule:

$$\texttt{assert(!x)}: \quad [\![v]\!] = \begin{cases} \sigma[\texttt{x} \mapsto \alpha_{\text{a}}(\{0\})] & \text{if } 0 \in \gamma_{\text{a}}(\sigma(\texttt{x})) \\ \bot & \text{otherwise} \end{cases}$$
$$\text{where } \sigma = \mathit{JOIN}(v)$$

This rule soundly models the fact that only executions where `x` has the value 0 satisfy the branch condition. However, with this change of the analysis, the abstract value of `x` at the `output` statement is now `0-`, so the analysis is no longer able to conclude that the output value is definitely not a negative number. In other words, even though the analysis has apparently become more precise by adding a simple form of control sensitivity, it is less precise for some programs, which may be counterintuitive.

Exercise 12.23: Continuing Exercise 12.22, let us add a lattice element `0`, below `0-` and `0+` and above $\bot$, with $\gamma_{\text{a}}(\texttt{0}) = \{0\}$. Show that with this modification, an abstraction function α_{a} exists such that α_{a} and γ_{a} form a Galois connection.

Exercise 12.24: In interval analysis (Section 6.1) we use the domain $\mathcal{P}(\mathbb{Z})$ representing sets of concrete values and the domain $\mathit{Interval}$ representing abstract values. What are the abstraction function and the concretization function between these domains? Do they form a Galois connection?

For the sign analysis (with the ordinary Sign lattice) we can also specify the connection between concrete values in $\mathbb{Z}$ and abstract values in Sign using a *representation function*, $\beta \colon \mathbb{Z} \to \mathit{Sign}$:

$$\beta(d) = \begin{cases} + & \text{if } d > 0 \\ - & \text{if } d < 0 \\ \texttt{0} & \text{if } d = 0 \end{cases}$$

The abstraction function α_{a} can then be induced from the representation function:

$$\alpha_{\text{a}}(D) = \bigsqcup \{\beta(d) \mid d \in D\}$$

Exercise 12.25: Show that this definition of α_{a} coincides with the one from page 169.

Intuitively, for a given set of concrete values, this construction of α_{a} picks the most precise abstract element that safely approximates the representation of each of the concrete values. Often it is more natural to specify a representation function than an abstraction function or a concretization function. The following exercise shows that this works whenever a suitable representation function can be defined.

Exercise 12.26: Assume $\beta\colon V \to L$ where V is a set and L is a complete lattice. Define $\alpha\colon \mathcal{P}(V) \to L$ and $\gamma\colon L \to \mathcal{P}(V)$ by $\alpha(D) = \bigsqcup\{\beta(d) \mid d \in D\}$ and $\gamma(x) = \{v \in V \mid \beta(v) \sqsubseteq x\}$. Prove that α and γ form a Galois connection.

The next two exercises show that we can build Galois connections using product lattices and map lattices (see Section 4.3).

Exercise 12.27: Assume L_1, L_2, L_1', and L_2' are complete lattices, $\alpha\colon L_1 \to L_2$ and $\gamma\colon L_2 \to L_1$ form a Galois connection, and $\alpha'\colon L_1' \to L_2'$ and $\gamma'\colon L_2' \to L_1'$ form a Galois connection. Define $\alpha''\colon L_1 \times L_1' \to L_2 \times L_2'$ and $\gamma''\colon L_2 \times L_2' \to L_1 \times L_1'$ by $\alpha''(x, x') = (\alpha(x), \alpha'(x'))$ and $\gamma''(y, y') = (\gamma(y), \gamma'(y'))$ for all $x \in L_1$, $x' \in L_1'$, $y \in L_2$, $y' \in L_2'$. Prove that α'' and γ'' form a Galois connection between the product lattices $L_1 \times L_1'$ and $L_2 \times L_2'$.

Exercise 12.28: Assume L_1 and L_2 are complete lattices, S is a set, and $\alpha\colon L_1 \to L_2$ and $\gamma\colon L_2 \to L_1$ form a Galois connection. Define $\alpha'\colon (S \to L_1) \to (S \to L_2)$ and $\gamma'\colon (S \to L_2) \to (S \to L_1)$ by $\alpha'(x)(s) = \alpha(x(s))$ and $\gamma'(y)(s) = \gamma(y(s))$ for all $x\colon S \to L_1$, $y\colon S \to L_2$, and $s \in S$. Prove that α' and γ' form a Galois connection between the map lattices $S \to L_1$ and $S \to L_2$.

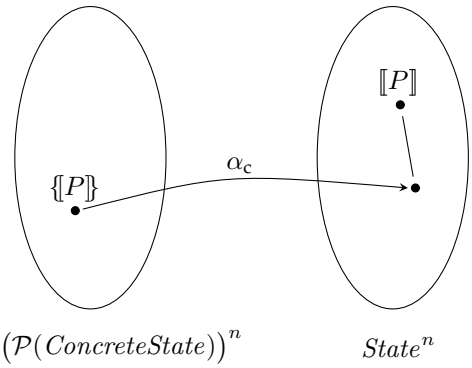
Exercise 12.29: Use the results of Exercises 12.27 and 12.28 to give a simpler solution to Exercise 12.13.

12.3 Soundness

We are now in a position to formally define what we mean by soundness of an analysis. Let $\alpha\colon L_1 \to L_2$ be an abstraction function where L_1 is the lattice for a collecting semantics and L_2 is the lattice for an analysis. As an example, $\alpha_{\mathsf{c}}\colon \big(\mathcal{P}(\mathit{ConcreteState})\big)^n \to \mathit{State}^n$ defined in Section 12.2 is such a function for the sign analysis. An analysis is *sound* with respect to the semantics and the abstraction function for a given program P if the following property holds:

$$\alpha(\{\!|P|\!\}) \sqsubseteq [\![P]\!]$$

In other words, soundness means that the analysis result over-approximates the abstraction of the semantics of the program. For the sign analysis, the property can be illustrated like this:



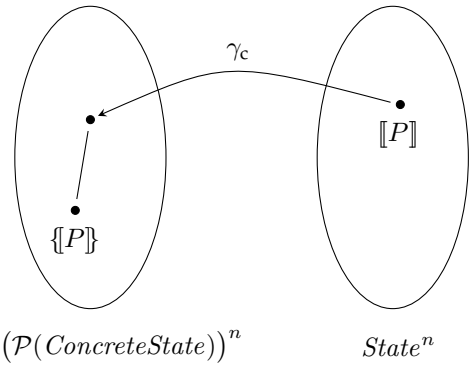
For the simple TIP example program considered in Section 12.1, the soundness property is indeed satisfied (here showing the information just for the program point immediately after the `c = a - b` statement):

$$\begin{array}{l}\alpha_{\mathsf{c}}(\ldots, \{\![\texttt{c = a - b}]\!\}, \ldots) = \\ \alpha_{\mathsf{c}}(\ldots, [\mathsf{a} \mapsto 42, \mathsf{b} \mapsto 87, \mathsf{c} \mapsto -45], \ldots) \sqsubseteq \\ (\ldots, [\mathsf{a} \mapsto +, \mathsf{b} \mapsto +, \mathsf{c} \mapsto \top], \ldots) = \\ (\ldots, [\![\texttt{c = a - b}]\!], \ldots)\end{array}$$

If we specify the relation between the two domains using concretization functions instead of using abstraction functions, we may dually define soundness as the property that the concretization of the analysis result over-approximates the semantics of the program:

$$\{\![P]\!\} \sqsubseteq \gamma([\![P]\!])$$

For the sign analysis, which uses the concretization function $\gamma_{\mathsf{c}} \colon State^n \to \big(\mathcal{P}(ConcreteState)\big)^n$, this property can be illustrated as follows:



Exercise 12.30: Show that if α and γ form a Galois connection, then the two definitions of soundness stated above are equivalent.
(Hint: see Exercise 12.17.)

We often use the term soundness of analyses without mentioning specific programs. An analysis is sound if it is sound for every program.

In Section 12.2 we established the relations between the concrete domains from the semantics and the abstract domains from the analysis. To prove that an analysis is sound, we also need to relate the semantic constraints with the analysis constraints. In the following, we outline the steps involved in such a proof for the sign analysis. We assume that the relations between the domains are specified using abstraction functions; if instead using concretization functions, the properties that need to be established are dual, similar to the above definition of soundness based on concretization functions.

First, $eval$ is a sound abstraction of $ceval$, in the sense that the following property holds for every expression E and every set of concrete states $R \subseteq ConcreteState$:

$$\alpha_{\mathsf{a}}(ceval(R, E)) \sqsubseteq eval(\alpha_{\mathsf{b}}(R), E)$$

Exercise 12.31: Prove that $eval$ is a sound abstraction of $ceval$, in the sense defined above.

(Hint: Use induction in the structure of the TIP expression. As part of the proof, you need to show that each abstract operator is a sound abstraction of the corresponding concrete operator, for example for the addition operator: $\alpha_{\mathsf{a}}(\{z_1 + z_2 \mid z_1 \in D_1 \ \wedge \ z_2 \in D_2\}) \sqsubseteq \alpha_{\mathsf{a}}(D_1) \mathbin{\widehat{+}} \alpha_{\mathsf{a}}(D_2)$ for all sets $D_1, D_2 \subseteq \mathbb{Z}$.)

The $succ$ function is a sound abstraction of $csucc$:

$$csucc(R, v) \subseteq succ(v) \text{ for any } R \subseteq ConcreteState$$

and $JOIN$ (defined on page 52) is a sound abstraction of $CJOIN$, meaning that

$$\alpha_{\mathsf{b}}(CJOIN(v)) \sqsubseteq JOIN(v)$$

for every CFG node $v \in Node$ whenever the constraint variables that are used in the definitions of $JOIN$ and $CJOIN$ satisfy $\alpha_{\mathsf{b}}(\{\!|w|\!\}) \sqsubseteq [\![w]\!]$ for all $w \in Node$.

Exercise 12.32: Prove that $succ$ is a sound abstraction of $csucc$ and that $JOIN$ is a sound abstraction of $CJOIN$, in the sense defined above.

Let $cf_v \colon \big(\mathcal{P}(ConcreteState)\big)^n \to \mathcal{P}(ConcreteState)$ and $af_v \colon State^n \to State$ denote v's constraint function from the semantics and the analysis, respectively, for every CFG node v. For example, if v represents an assignment statement X = E, we have:

$$cf_v(\{\!|v_1|\!\}, \ldots, \{\!|v_n|\!\}) = \big\{\rho[X \mapsto z] \;\big|\; \rho \in CJOIN(v) \ \wedge \ z \in ceval(\rho, E)\big\}$$
$$af_v([\![v_1]\!], \ldots, [\![v_n]\!]) = \sigma[X \mapsto eval(\sigma, E)] \text{ where } \sigma = JOIN(v)$$

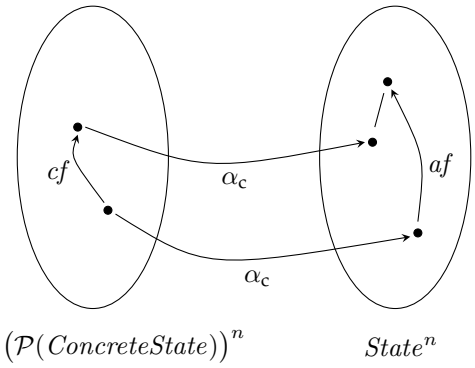
The function af_v is a sound abstraction of cf_v if the following property holds for all $R_1, \ldots, R_n \subseteq ConcreteState$:

$$\alpha_{\mathsf{b}}(cf_v(R_1, \ldots, R_n)) \sqsubseteq af_v(\alpha_{\mathsf{b}}(R_1), \ldots, \alpha_{\mathsf{b}}(R_n))$$

If we consider the combined constraint functions for the entire program, $cf(\{\!\{v_1\}\!\},\ldots,\{\!\{v_n\}\!\}) = \big(cf_{v_1}(\{\!\{v_1\}\!\},\ldots,\{\!\{v_n\}\!\}),\ldots,cf_{v_n}(\{\!\{v_1\}\!\},\ldots,\{\!\{v_n\}\!\})\big)$ and $af([\![v_1]\!],\ldots,[\![v_n]\!]) = \big(af_{v_1}([\![v_1]\!],\ldots,[\![v_n]\!]),\ldots,af_{v_n}([\![v_1]\!],\ldots,[\![v_n]\!])\big)$ then af being a sound abstraction of cf means that

$$\alpha_{\mathrm{c}}(cf(R_1,\ldots,R_n)) \sqsubseteq af(\alpha_{\mathrm{c}}(R_1,\ldots,R_n))$$

which can be illustrated like this:



Exercise 12.33: Prove that for each kind of CFG node v, the sign analysis constraint af_v is a sound abstraction of the semantic constraint cf_v. (The most interesting case is the one where v is an assignment node.) Then use that result to prove that af is a sound abstraction of cf.

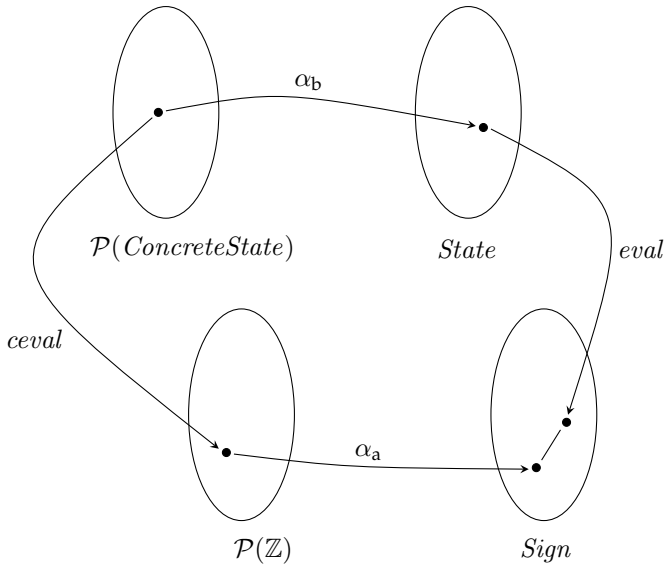
We can provide a general definition of soundness of abstractions, covering all the variants above, as follows. Assume L_1 and L_1' are lattices used in a collecting semantics and L_2 and L_2' are lattices used for an analysis. Let $\alpha\colon L_1 \to L_2$ and $\alpha'\colon L_1' \to L_2'$ be abstraction functions, and let $\gamma\colon L_2 \to L_1$ and $\gamma'\colon L_2' \to L_1'$ be concretization functions such that α, γ, α', and γ' form two Galois connections. Consider two monotone functions $cg\colon L_1 \to L_1'$ and $ag\colon L_2 \to L_2'$. We say that ag is a *sound* abstraction of cg if the following property holds:

$$\alpha' \circ cg \sqsubseteq ag \circ \alpha$$

The following exercise provides an alternative definition of soundness that is based on concretization functions instead of abstraction functions.

Exercise 12.34: Prove that if α, α', γ, and γ' form two Galois connections as above, then $\alpha' \circ cg \sqsubseteq ag \circ \alpha$ holds if and only if $cg \circ \gamma \sqsubseteq \gamma' \circ ag$ holds.

As an example for our sign analysis, $eval$ being a sound abstraction of $ceval$ means that $\alpha_{\mathrm{a}} \circ ceval \sqsubseteq eval \circ \alpha_{\mathrm{b}}$ (for a fixed expression), which can be illustrated as follows:



Intuitively, when starting from a set of concrete states, if we first abstract the states and then evaluate abstractly with $eval$ we get an abstract value that over-approximates the one we get if we first evaluate concretely with $ceval$ and then abstract the values.

With the result of Exercise 12.33, it follows that the sign analysis is sound with respect to the semantics and the abstraction/concretization functions, as shown next.

Recall that the analysis result for a given program P is computed as $[\![P]\!] = \mathit{lfp}(af)$ (if not using widening, which we discuss later in this section). By the fixed-point theorems from Section 12.1, the semantics of P is similarly given by $\{\![P]\!\} = \mathit{lfp}(cf)$. According to the definition of soundness, we thus need to show that $\alpha(\mathit{lfp}(cf)) \sqsubseteq \mathit{lfp}(af)$ (if the connection between the lattices is specified using an abstraction function α) or $\mathit{lfp}(cf) \sqsubseteq \gamma(\mathit{lfp}(af))$ (if using a concretization function γ, cf. Exercise 12.17).

The central result we need is the following *soundness theorem*:

> If L_1 and L_2 are complete lattices with a concretization function $\gamma\colon L_2 \to L_1$, $cf\colon L_1 \to L_1$ and $af\colon L_2 \to L_2$ are monotone, and af is a sound abstraction of cf with respect to γ, i.e., $cf \circ \gamma \sqsubseteq \gamma \circ af$, then $\mathit{lfp}(cf) \sqsubseteq \gamma(\mathit{lfp}(af))$.

Applying this theorem to the sign analysis amounts to setting $L_1 = \big(\mathcal{P}(ConcreteState)\big)^n$, $L_2 = State^n$, and $\gamma = \gamma_{\mathsf{c}}$. The functions cf and af are the constraint functions for the semantics and the analysis, respectively, of the program P being analyzed.

Proving this theorem is straightforward thanks to Tarski's fixed point theorem (see page 81). Assume the conditions for L_1, L_2, γ, cf, and af are satisfied.

We know from Tarski's fixed point theorem that $lfp(cf)$ and $lfp(af)$ are well-defined. As $lfp(af)$ is a fixed point of af we have $af(lfp(af)) = lfp(af)$, so $\gamma(af(lfp(af))) = \gamma(lfp(af))$. Since af is a sound abstraction of cf we then have $cf(\gamma(lfp(af))) \sqsubseteq \gamma(lfp(af))$. This means that $\gamma(lfp(af)) \in \{x \in L_1 \mid cf(x) \sqsubseteq x\}$, so by Tarski's fixed point theorem, $lfp(cf) \sqsubseteq \gamma(lfp(af))$.

To summarize, a general recipe for specifying and proving soundness of an analysis consists of the following steps:

1. Specify the analysis, i.e. the analysis lattice (a complete lattice) and the constraint generation rules, and check that all the analysis constraint functions are monotone (as we did for the sign analysis example in Section 5.1).

2. Specify the collecting semantics, and check that the semantic constraint functions are monotone (as we did for the sign analysis example in Section 12.1). The collecting semantics must capture the desired aspects of concrete execution, such that it formalizes the ideal properties that the analysis is intended to approximate. For the sign analysis example, we designed the collecting semantics such that it collects the reachable states for every program point; other analyses may need other kinds of collecting semantics (see Section 12.6).

3. Establish the connection between the semantic lattice and the analysis lattice (as we did for the sign analysis example in Section 12.2), either by an abstraction function or by a concretization function. Alternatively, one may use a representation function (Exercise 12.26) and induce the abstraction and concretization functions. For the sign analysis example, the lattices are defined in three layers, leading to the definitions of α_{a}, α_{b}, α_{c} and γ_{a}, γ_{b}, γ_{c}. By ensuring that the Galois connection property is satisfied, it is only a matter of taste whether one chooses to specify this connection using abstraction functions or using concretization functions, thanks to the Galois connection dualities we have seen in Section 12.2. (However, notice that the soundness theorem does not require the presence of a Galois connection, but merely that the abstraction and concretization functions are monotone.)

4. Show that each constituent of the analysis constraints is a sound abstraction of the corresponding constituent of the semantic constraints (with respect to the concretization function), for all programs (as we did for the sign analysis example in Exercises 12.31, 12.32, and 12.33).

5. Soundness of the analysis then follows from the soundness theorem stated above.

For analyses that use widening, we know from Exercise 6.9 that the analysis result is always a safe approximation of the least solution to the analysis constraints. This means that the soundness theorem can also be used for such analyses.

Exercise 12.35: Prove that the interval analysis (Section 6.1) with widening (using the definition of ∇ from page 86) is sound with respect to the collecting semantics from Section 12.1.

Proving soundness of realistic analyses for real-world programming languages is a major endeavor [JLB+15]. A pragmatic light-weight alternative is *soundness testing* [AMN17], which is the process of running a given program concretely a number of times, with as high coverage as possible, and testing that all observed runtime facts are over-approximated by the static analysis result.

12.4 Optimality

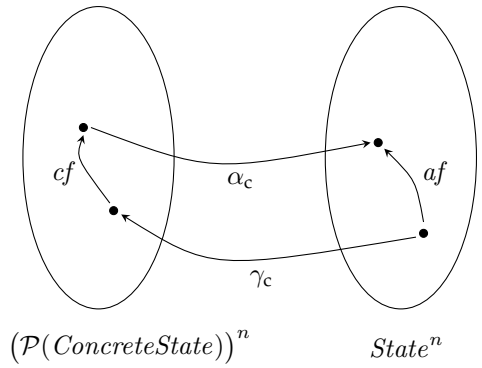
Assume we are developing a new analysis, and that we have chosen an analysis lattice and the rules for generating analysis constraints for the various programming language constructs. To enable formal reasoning about the soundness and precision of the analysis, we have also provided a suitable collecting semantics for the programming language (as in Section 12.1) and abstraction/concretization functions that define the meaning of the analysis lattice elements (as in Section 12.2). Furthermore, assume we have proven that the analysis is sound using the approach from Section 12.3. We may now ask: *Are our analysis constraints as precise as possible* (*yet sound*)*, relative to the chosen analysis lattice?*

As in the previous section, let $\alpha\colon L_1 \to L_2$ be an abstraction function where L_1 is the lattice for a collecting semantics and L_2 is the lattice for an analysis, such that α and γ form a Galois connection, and consider two functions $cf\colon L_1 \to L_1$ and $af\colon L_2 \to L_2$ that represent, respectively, the semantic constraints and the analysis constraints for a given program. We say that af is the *optimal*[6] abstraction of cf if

$$af = \alpha \circ cf \circ \gamma$$

(which can also be written: $af(b) = \alpha(cf(\gamma(b)))$ for all $b \in L_2$). Using the lattices and abstraction/concretization functions from the sign analysis example, this property can be illustrated as follows.

[6] In the literature on abstract interpretation, the term "best" is sometimes used instead of "optimal".



(Compare this with the illustration of soundness from page 179.) To see that $\alpha \circ cf \circ \gamma$ is indeed the most precise monotone function that is a sound abstraction of cf, assume $g\colon L_2 \to L_2$ is some monotone function that is a sound abstraction of cf, that is, $\alpha(cf(a)) \sqsubseteq g(\alpha(a))$ for all $a \in L_1$. Then for all $a' \in L_2$, $\alpha(cf(\gamma(a'))) \sqsubseteq g(\alpha(\gamma(a'))) \sqsubseteq g(a')$. The last inequality holds because $\alpha \circ \gamma$ is reductive and g is monotone. Thus, $\alpha \circ cf \circ \gamma \sqsubseteq g$, meaning that $\alpha \circ cf \circ \gamma$ is the most precise such function.

Notice what the optimality condition tells us: We can obtain the best possible (i.e., most precise yet sound) abstraction of *cf* for a given analysis lattice element y by first concretizing y, then applying the semantic function cf, and finally abstracting. Unfortunately this observation does not automatically give us practical algorithms for computing optimal abstraction, but it enables us to reason about the precision of our manually specified analysis constraints.

The above definition of optimality focuses on af and cf, but it can be generalized to all levels of the analysis as follows. Assume L_1 and L_1' are lattices used in a collecting semantics and L_2 and L_2' are lattices used for an analysis. Let $\alpha\colon L_1 \to L_2$ and $\alpha'\colon L_1' \to L_2'$ be abstraction functions, and let $\gamma\colon L_2 \to L_1$ and $\gamma'\colon L_2' \to L_1'$ be concretization functions such that α, γ, α', and γ' form two Galois connections. Consider two functions $cg\colon L_1 \to L_1'$ and $ag\colon L_2 \to L_2'$. We say that ag is the *optimal* abstraction of cg if the following property holds:

$$ag = \alpha' \circ cg \circ \gamma$$

Let us look at some examples from the sign analysis. First, it is easy to see that our definition of abstract multiplication $\widehat{*}$ (page 53) is the optimal abstraction of concrete multiplication, denoted "$\cdot$":

$$s_1 \mathbin{\widehat{*}} s_2 = \alpha_{\mathsf{a}}\big(\gamma_{\mathsf{a}}(s_1) \cdot \gamma_{\mathsf{a}}(s_2)\big)$$

for any $s_1, s_2 \in \mathit{Sign}$, where we overload the $\cdot$ operator to work on sets of integers: $D_1 \cdot D_2 = \{z_1 \cdot z_2 \mid z_1 \in D_1 \ \wedge \ z_2 \in D_2\}$ for any $D_1, D_2 \subseteq \mathbb{Z}$.

Exercise 12.36: Prove that all the abstract operators ($\widehat{+}$, $\widehat{-}$, $\widehat{>}$, etc.) defined in Section 5.1 are optimal abstractions of their concrete counterparts. (This exercise can be seen as a formal version of Exercise 5.4.)

Despite the result of the previous exercise, the $eval$ function from Section 5.1 is *not* the optimal abstraction of $ceval$. Here is a simple counterexample: Let $\sigma \in State$ such that $\sigma(\mathtt{x}) = \top$ and consider the TIP expression `x - x`. We then have

$$eval(\sigma, \mathtt{x - x}) = \top$$

while

$$\alpha_{\mathrm{a}}\big(ceval(\gamma_{\mathrm{b}}(\sigma), \mathtt{x - x})\big) = \mathtt{0}$$

(This is essentially the same observation as the one in Exercise 5.9, but this time stated more formally.) Interestingly, defining the $eval$ function inductively and compositionally in terms of optimal abstractions does not make the function itself optimal.

Exercise 12.37: Assume we only work with normalized TIP programs (as in Exercise 2.2). Give an alternative computable definition of $eval$ for sign analysis (i.e., an algorithm for computing $eval(\sigma, E)$ for any abstract state σ and normalized TIP expression E), such that $eval$ is the optimal abstraction of $ceval$.

Exercise 12.38: Is it possible to solve Exercise 12.37 without the normalization assumption?

Recall the definition of the abstract operators in interval analysis from Chapter 6 (page 80):

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

where $[l_1, h_1], [l_2, h_2] \in Interval$. Notice that this is by construction an optimal abstraction. As pointed out in Exercise 6.2 it is not immediately implementable, so a non-optimal (but still sound) alternative may be preferred in practice.

Exercise 12.39: Which of the other abstractions used in interval analysis (Section 6.1) are optimal?

To be able to reason about optimality of the abstractions used in, for example, live variables analysis or reaching definitions analysis, we first need a style of collecting semantics that is suitable for those analyses, which we return to in Section 12.6.

12.5 Completeness

As usual in logics, the dual of soundness is completeness. In Section 12.3 we defined soundness of an analysis for a program P as the property $\alpha(\{\!|P|\!\}) \sqsubseteq [\![P]\!]$. Consequently, it is natural to define that an analysis is *complete* for P if the following property holds:

$$[\![P]\!] \sqsubseteq \alpha(\{\!|P|\!\})$$

An analysis is complete if it is complete for all programs. If an analysis is both sound and complete for P we have $\alpha(\{\!|P|\!\}) = [\![P]\!]$.[7]

In Section 12.4 we studied the notion of optimality of abstractions, motivated by the interest in defining analysis constraints to be as precise as possible, relative to the chosen analysis lattice. We can similarly ask, is the analysis result $[\![P]\!]$ for a program P as precise as possible for the currently used analysis lattice? Stated more formally, the question is whether $\alpha(\{\!|P|\!\}) = [\![P]\!]$ holds; in other words, this property coincides with the analysis being sound and complete for P.

Even if we manage to solve Exercise 12.37 and obtain an optimal definition of $eval$ for sign analysis, the analysis is not sound and complete for all (normalized) programs, as demonstrated by the following counterexample:

```
x = input;
y = x;
z = x - y;
```

Let σ denote the abstract state after the statement `y = x` such that $\sigma(\mathtt{x}) = \sigma(\mathtt{y}) = \top$. Any sound abstraction of the semantics of the single statement `z = x - y` will result in an abstract state that maps `z` to $\top$, but the answer `0` would be more precise and still sound in the analysis result for the final program point. Intuitively, the analysis does not know about the correlation between `x` and `y`.

For this specific example program, we could in principle improve analysis precision by changing the constraint generation rules to recognize the special pattern consisting of the statement `y = x` followed by `z = x - y`. Instead of such an ad hoc approach to gain precision, relational analysis (see Chapter 7) is usually a more viable solution.

In Section 12.3 we observed (Exercise 12.30) that analysis soundness could equivalently be defined as the property $\{\!|P|\!\} \sqsubseteq \gamma([\![P]\!])$. However, a similar equivalence does *not* hold for completeness, as shown by the following two exercises.

Exercise 12.40: We know from Exercise 12.17 that if α and γ form a Galois connection then $\alpha(x) \sqsubseteq y \iff x \sqsubseteq \gamma(y)$ for all x, y. Prove (by showing a counterexample) that the converse property does not hold, i.e. $\alpha(x) \sqsupseteq y \not\Longleftrightarrow x \sqsupseteq \gamma(y)$.
(Hint: consider α_a and γ_a from the sign analysis.)

Exercise 12.41: Give an example of a program P such that $[\![P]\!] \sqsubseteq \alpha(\{\!|P|\!\})$ and $\gamma([\![P]\!]) \not\sqsubseteq \{\!|P|\!\}$ for sign analysis.

Thus, $\{\!|P|\!\} = \gamma([\![P]\!])$ is a (much) stronger property than $\alpha(\{\!|P|\!\}) = [\![P]\!]$. If $\{\!|P|\!\} = \gamma([\![P]\!])$ is satisfied, the analysis captures exactly the semantics of P without any approximation; we say that the analysis is *exact* for P. Every

[7] The literature on abstract interpretation often uses the term "complete" for what we call "sound and complete", by working under the assumption of sound analyses.

nontrivial abstraction loses information and therefore no interesting analysis is exact for all programs.[8] (Still, the property may hold for *some* programs.)

Exercise 12.42:
(a) Explain why the sign analysis is both complete and exact for this program:

```
var x;
x = 0;
```

(b) Explain why the sign analysis is complete but not exact for this program:

```
var x;
x = 1;
```

Having established a notion of analysis completeness (and a less interesting notion of analysis exactness), we proceed by defining a notion of completeness of the individual abstractions used in an analysis, to understand where imprecision may arise.

As in the preceding sections, assume L_1 and L'_1 are lattices used in a collecting semantics and L_2 and L'_2 are lattices used for an analysis. Let $\alpha\colon L_1 \to L_2$ and $\alpha'\colon L'_1 \to L'_2$ be abstraction functions, and let $\gamma\colon L_2 \to L_1$ and $\gamma'\colon L'_2 \to L'_1$ be concretization functions such that α, γ, α', and γ' form two Galois connections. Consider two functions $cg\colon L_1 \to L'_1$ and $ag\colon L_2 \to L'_2$. We say that ag is a *complete* abstraction of cg if the following property holds:

$$ag \circ \alpha \sqsubseteq \alpha' \circ cg$$

(Compare this with the definition of soundness of abstractions from page 179.)

Again, let us consider sign analysis as example. In Section 12.4 we saw that abstract multiplication is optimal. In fact, it is also complete:

$$\alpha_{\mathrm{a}}(D_1)\,\widehat{*}\,\alpha_{\mathrm{a}}(D_2) \sqsubseteq \alpha_{\mathrm{a}}(D_1 \cdot D_2)$$

for any $D_1, D_2 \subseteq \mathbb{Z}$. This tells us that the analysis, perhaps surprisingly, never loses any precision at multiplications.

For abstract addition, the situation is different, as shown in the next exercise.

Exercise 12.43: Abstract addition in sign analysis is *not* complete. Give an example of two sets $D_1, D_2 \subseteq \mathbb{Z}$ where $\alpha_{\mathrm{a}}(D_1)\,\widehat{+}\,\alpha_{\mathrm{a}}(D_2) \not\sqsubseteq \alpha_{\mathrm{a}}(D_1 + D_2)$. (We here overload the $+$ operator to work on sets of integers, like we did earlier for multiplication.)

Is it possible to change the definition of abstract addition to make it sound and complete (without changing the analysis lattice)?

[8] These observations show that we could in principle have chosen define the concept of completeness using the concretization function γ instead of using the abstraction function α, but that would have been much less useful.

Exercise 12.44: For which of the operators `-`, `/`, `>`, and `==` is sign analysis complete? What about `input`?

Notice that least upper bound in an analysis lattice L_2 is always a sound and complete abstraction of least upper bound in the corresponding collecting semantics lattice L_1 whenever there is a Galois connection between L_1 and L_2:

$$\alpha(\bigsqcup A) = \bigsqcup_{a \in A} \alpha(a) \text{ where } A \subseteq L_1$$

(This result follows directly from Exercise 12.18.) Intuitively this means, perhaps surprisingly, that joining abstract information, for example when the branches merge after an `if` statement, is never to blame for precision losses.

Exercise 12.45: In sign analysis, is the analysis constraint function for assignments $af_{\mathtt{X=E}}$ a complete abstraction of the corresponding semantic constraint function $cf_{\mathtt{X=E}}$, given that E is an expression for which $eval$ is complete?

For abstractions that are sound, completeness implies optimality (but not vice versa, cf. exercises 12.36 and 12.43):

Exercise 12.46: Prove that if ag is sound and complete with respect to cg, it is also optimal.

We have seen in Section 12.4 that for any Galois connection, there exists an optimal (albeit often non-computable) abstraction of every concrete operation. The same does not hold for soundness and completeness, as shown in the following exercise.

Exercise 12.47: Prove that there exists a sound and complete abstraction ag of a given concrete operation cg if and only if the optimal abstraction of cg is sound and complete.

(We have seen examples of abstractions that are optimal but not sound and complete, so this result implies that sound and complete abstractions do not always exist.)

The following exercise provides a "soundness and completeness theorem", as a variant of the soundness theorem from page 180.

Exercise 12.48: Prove that if af is sound and complete with respect to cf then $\alpha(\{\![P]\!\}) = [\![P]\!]$, where cf is the semantic constraint function and af is the analysis constraint function for a given program P, and α is the abstraction function.

This theorem relies on the fact that sound and complete abstractions are closed under composition, unlike optimal abstractions, as seen in the following

exercise.

Exercise 12.49:

(a) Prove that if af is sound and complete with respect to cf then $af \circ af$ is sound and complete with respect to $cf \circ cf$. (Hint: see your solution to Exercise 12.48.)

(b) Prove that the following statement does *not* hold in general: If af is the optimal abstraction of cf then $af \circ af$ is the optimal abstraction of $cf \circ cf$. (Hint: consider the example program on page 185.)

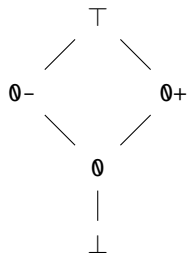
Since $[\![P]\!] = \bigsqcup_{i \geq 0} af^i(\bot)$ by the fixed-point theorem, part (b) of this result shows that even when the analysis constraint af for a given program P is the most precise possible (relative to the chosen analysis lattice), the analysis result $[\![P]\!]$ may not be the most precise possible (that can be expressed in the analysis lattice). In contrast, part (a) is the central property in the proof of the soundness and completeness theorem in Exercise 12.48.

Exercise 12.50: Which of the abstractions used in interval analysis (Section 6.1) are complete? In particular, is abstract addition complete?

Abstractions that are incomplete may be complete in some situations; for example, abstract addition in sign analysis is not complete in general (Exercise 12.43), but it is complete in situations where, for example, both arguments are positive values. For this reason, even though few analyses are sound and complete for *all* programs, many analyses are sound and complete for *some* programs or program fragments.

Exercise 12.51: Prove that abstract addition in sign analysis is complete if both arguments are positive values. That is, show that $\alpha_{\mathrm{a}}(D_1) \widehat{+} \alpha_{\mathrm{a}}(D_2) \sqsubseteq \alpha_{\mathrm{a}}(D_1 + D_2)$ for all $D_1, D_2 \subseteq \{1, 2, 3, \dots\}$.

There are not many programs for which our simple sign analysis is complete and gives a nontrivial analysis result, so to be able to demonstrate how these observations may be useful, let us modify the analysis to use the following slightly different lattice instead of the usual *Sign* lattice from Section 5.1.



The meaning of the elements is expressed by this concretization function γ_{a}:

$$\gamma_{\mathrm{a}}(s) = \begin{cases} \emptyset & \text{if } s = \bot \\ \{0\} & \text{if } s = \texttt{0} \\ \{0, 1, 2, 3, \dots\} & \text{if } s = \texttt{0+} \\ \{0, -1, -2, -3, \dots\} & \text{if } s = \texttt{0-} \\ \mathbb{Z} & \text{if } s = \top \end{cases}$$

From Exercise 12.23 we know that an abstraction function α_{a} exists such that α_{a} and γ_{a} form a Galois connection.

Exercise 12.52: Give a definition of *eval* that is optimal for expressions of the form `t * t` where `t` is any program variable (recall Exercise 12.37).

The remaining abstract operators can be defined similarly, and the rest of the *eval* function and other analysis constraints can be reused from the ordinary sign analysis.

The modified sign analysis concludes that output is `0+` for the following small program:

```
x1 = input;
x2 = input;
y1 = x1 * x1;
y2 = x2 * x2;
output y1 + y2;
```

Exercise 12.53: Explain why the modified sign analysis is sound and complete for this program.

Assume the analysis is built to raise an alarm if the output of the analyzed program is a negative value for some input. In this case, it will not raise an alarm for this program, and because we know the analysis is sound (over-approximating all possible behaviors of the program), this must be the correct result. Now assume instead that the analysis is built to raise an alarm if the output of the analyzed program is a *positive* value for some input. In this case, it does raise an alarm, and because we know the analysis is complete for this program (though not for all programs), this is again the correct result – there must exist an execution of the program that outputs a positive value; we can trust that the alarm is not a false positive.

Exercise 12.54: Design a relational sign analysis that is sound and complete for the three-line program from page 185.

Exercises 12.50 and 12.54 might suggest that increasing analysis precision generally makes an analysis complete for more programs, but that is not the case: The trivial analysis that uses a one-element analysis lattice is sound and complete for all programs, but it is obviously useless because its abstraction discards all information about any analyzed program.

For further discussion about the notion of completeness, see Giacobazzi et al. [GRS00, GLR15].

12.6 Trace Semantics

The TIP semantics presented in Section 12.1 is called a *reachable states* collecting semantics, because for each program point it collects the set of states that are possible when program execution reaches that point for some input. As we have seen, this precisely captures the meaning of TIP programs in a way that allows us to prove soundness of, for example, sign analysis. For other analyses, however, the reachable states collecting semantics is insufficient because it does not capture all the information about how TIP programs execute. As a trivial example, for the program

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

the reachable states collecting semantics will only tell us that the set of states at the function entry and the set of states at the function exit are both $\{[\mathtt{x} \mapsto z] \mid z \in \mathbb{Z}\}$, but it does not tell us that the return value is always the same as the input value. In other words, the reachable states collecting semantics does not provide information about how one state at a program point is related to states at other program points. To capture such aspects of TIP program execution, we can instead use a *trace semantics* that expresses the meaning of a TIP program as the set of traces that can appear when the program runs. A *trace* is a finite sequence of pairs of program points (represented as CFG nodes) and states:[9]

$$Trace = (Node \times ConcreteState)^*$$

We first define the semantics of single CFG nodes as functions from concrete states to sets of concrete states (as concrete counterparts to the transfer functions from Section 5.10): $ct_v : ConcreteState \to \mathcal{P}(ConcreteState)$.

For assignment nodes, ct_v can be defined as follows:

$$ct_{X=E}(\rho) = \{\rho[X \mapsto z] \mid z \in ceval(\rho, E)\}$$

The semantics of variable declaration nodes can be defined similarly. All other kinds of nodes do not change the state:

$$ct_v(\rho) = \{\rho\}$$

The trace semantics of a program P is a set of finite traces, written $\langle\!|P|\!\rangle \in \mathcal{P}(Trace)$. We can define $\langle\!|P|\!\rangle$ as the set of finite traces that start at the program entry point and in each step proceed according to the CFG. (We do not require

[9] For a set X, the Kleene star operation X^* denotes the set of all finite sequences of elements from X, including the empty sequence.

that the traces reach the program exit.) More formally, if `main` has parameters $x_1, \ldots, x_n$, we define $\langle\!|P|\!\rangle$ as the least solution to the following two constraints (similarly to how we defined $[\![P]\!]$ and $\{\!|P|\!\}$ earlier):

$$(\mathit{entry}, [x_1 \mapsto z_1, \ldots, x_n \mapsto z_n]) \in \langle\!|P|\!\rangle \qquad \text{for all } z_1, \ldots, z_n \in \mathbb{Z}$$

$$\pi \cdot (v, \rho) \in \langle\!|P|\!\rangle \;\wedge\; v' \in \mathit{csucc}(\rho, v) \;\wedge\; \rho' \in \mathit{ct}_{v'}(\rho) \implies \pi \cdot (v, \rho) \cdot (v', \rho') \in \langle\!|P|\!\rangle$$

The first constraint says that the program entry point is reachable in every initial state that assigns arbitrary integer values to the parameters of `main`. The second constraint says that if the program has a trace that ends at node v in state ρ such that v' is a possible successor node and ρ' is a state we may get if executing v' from state ρ, then the trace that is extended with the extra pair (v', ρ') is also possible.

As an example, if P is the trivial program above, we have

$$\begin{aligned}\langle\!|P|\!\rangle = \{&(\mathit{entry}, [\mathtt{x} \mapsto 0]) \cdot (\mathtt{return\ x}, [\mathtt{x} \mapsto 0]) \cdot (\mathit{exit}, [\mathtt{x} \mapsto 0]),\\ &(\mathit{entry}, [\mathtt{x} \mapsto 1]) \cdot (\mathtt{return\ x}, [\mathtt{x} \mapsto 1]) \cdot (\mathit{exit}, [\mathtt{x} \mapsto 1]),\\ &\ldots\}\end{aligned}$$

which contains the information that the value of `x` is the same at the entry and the exit, in any execution.

Exercise 12.55: Describe the trace semantics of the program from page 164.

Exercise 12.56: Explain why it makes sense to define the trace semantics as the *least* solution to the constraints. (Hint: see the discussion for the reachable states collecting semantics on page 167.)

Interestingly, the relation between the reachable states collecting semantics and the trace semantics can be expressed as a Galois connection induced by an abstraction function

$$\alpha_{\mathrm{t}} \colon \mathcal{P}(\mathit{Trace}) \to \big(\mathcal{P}(\mathit{ConcreteState})\big)^n$$

defined by

$$\alpha_{\mathrm{t}}(T) = (R_1, \ldots, R_n)$$
$$\text{where } R_i = \{\rho \mid \cdots \cdot (v_i, \rho) \cdot \cdots \in T\} \text{ for each } i = 1, \ldots, n.$$

Intuitively, given a set of traces, α_{t} simply extracts the set of reachable states for each CFG node.

The set $\mathcal{P}(\mathit{Trace})$ forms a powerset lattice, and α_{t} is a complete join morphism, so by Exercise 12.21 we know that a concretization function γ_{t} exists such that α_{t} and γ_{t} form a Galois connection.

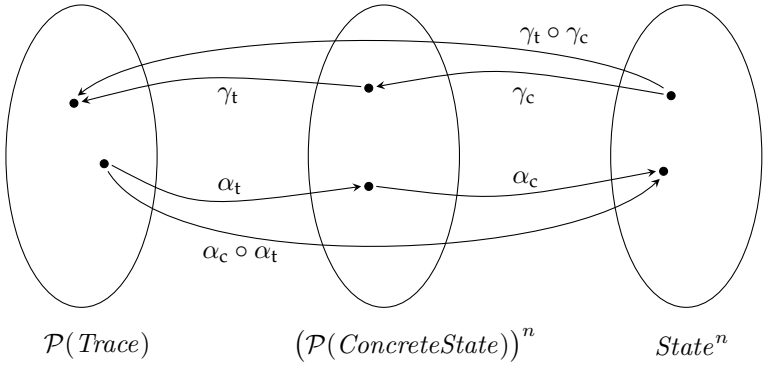
Exercise 12.57: Show that α_{t} (as defined above) is indeed a complete join morphism.

The existence of this Galois connection shows that the domain of the reachable states collecting semantics is in some sense an abstraction of the domain of the trace semantics.

The following exercise shows that composition of Galois connections leads to new Galois connections.

Exercise 12.58: Let $\alpha_1 : L_1 \to L_2$, $\gamma_1 : L_2 \to L_1$, $\alpha_2 : L_2 \to L_3$, and $\gamma_2 : L_3 \to L_2$. Assume both (α_1, γ_1) and (α_2, γ_2) are Galois connections. Prove that $(\alpha_2 \circ \alpha_1, \gamma_1 \circ \gamma_2)$ is then also a Galois connection.

In Section 12.2 we have established a Galois connection between the domain $\big(\mathcal{P}(\mathit{ConcreteState})\big)^n$ of the reachable states collecting semantics and the domain State^n of the sign analysis, and we have now also established a Galois connection between the domain $\mathcal{P}(\mathit{Trace})$ of the trace semantics and the domain $\big(\mathcal{P}(\mathit{ConcreteState})\big)^n$. Applying the result from Exercise 12.58 then gives us a Galois connection between $\mathcal{P}(\mathit{Trace})$ and State^n, which can be illustrated like this:



Exercise 12.59: Prove that the reachable states collecting semantics is sound with respect to the trace semantics. (Even though the collecting semantics is not a computable analysis, we can still apply the notion of soundness and the proof techniques from Section 12.3.)

To demonstrate the usefulness of trace semantics, let us consider how to prove soundness of the reaching definitions analysis from Section 5.7. Recall that an assignment *X* = *E* (called a *definition* of *X*) is *reaching* a program point v if in some execution of the program, v is reached, and the variable *X* was last assigned to at that assignment.

As a start, we define more formally what is meant by the set of reaching definitions at a given program point. We do this by first defining an *instrumented* trace semantics that extends the information in the traces with maps from variables to where the variables were last assigned to:

$$\mathit{Trace}_{RD} = (\mathit{Node} \times \mathit{ConcreteState} \times \mathit{ReachingDef})^*$$

$$ReachingDef = Var \hookrightarrow Node$$

Here, we have added a third component, a map from variables to CFG nodes, to each element of the traces. Define $\langle\!\langle P \rangle\!\rangle_{RD}$ as the least solution to the following two constraints, similarly to how we defined the ordinary trace semantics $\langle\!\langle P \rangle\!\rangle$ earlier, but capturing the reaching definitions information:

$$(\mathit{entry}, [x_1 \mapsto z_1, \ldots, x_n \mapsto z_n], []) \in \langle\!\langle P \rangle\!\rangle_{RD} \qquad \text{for all } z_1, \ldots, z_n \in \mathbb{Z}$$

$$\pi \cdot (v, \rho, \delta) \in \langle\!\langle P \rangle\!\rangle_{RD} \ \wedge\ v' \in csucc(\rho, v) \ \wedge\ \rho' \in ct_{v'}(\rho) \implies \pi \cdot (v, \rho, \delta) \cdot (v', \rho', \delta') \in \langle\!\langle P \rangle\!\rangle_{RD}$$

where

$$\delta' = \begin{cases} \delta[X \mapsto v'] & \text{if } v' \text{ is an assignment } X = E \\ \delta & \text{otherwise} \end{cases}$$

In each element of the traces, the first two components are exactly as before. The reaching definitions map in the third component is initially empty and is updated whenever an assignment is encountered.

We also define an abstraction function $\alpha_{RD} : \mathcal{P}(Trace_{RD}) \to (\mathcal{P}(Node))^n$ that extracts the sets of semantically reaching definitions from a set of instrumented traces for a program with n program points:

$$\alpha_{RD}(T) = (R_1, \ldots, R_n)$$

where

$$R_i = \{w \mid \cdots (v_i, \rho, \delta) \cdots \in T \text{ where } \delta(X) = w \text{ for some } \rho, \delta, X\}$$

for $i = 1, \ldots, n$. Intuitively, when execution is at a program point v_i, the δ function contains the definitions that reach v_i.

We can now state formally what it means for the reaching definitions analysis to be sound for a program P:

$$\alpha_{RD}(\langle\!\langle P \rangle\!\rangle_{RD}) \sqsubseteq [\![P]\!]_{RD}$$

where $[\![P]\!]_{RD}$ denotes the reaching definitions analysis result for P.

Exercise 12.60: Describe $\langle\!\langle P \rangle\!\rangle_{RD}$ and $[\![P]\!]_{RD}$ for P being the example program from Section 5.7.

Exercise 12.61: With this instrumented trace semantics, use the approach from Section 12.3 to prove that the reaching definitions analysis from Section 5.7 is sound.

Reasoning about other analyses generally requires other forms of instrumented semantics that capture the semantic properties of interest.

Exercise 12.62: Use the approach from Section 12.3 and an appropriate instrumented trace semantics to prove that the available expressions analysis from Section 5.5 is sound. (This is trickier than Exercise 12.61, because available expressions analysis is a "must" analysis!)

Exercise 12.63: Use the approach from Section 12.3 and an appropriate instrumented trace semantics to prove that the live variables analysis from Section 5.4 is sound. As part of this, you need to specify an appropriate collecting semantics that formally captures what it means for a variable to be live (see the informal definition in Section 5.4). (This is trickier than Exercise 12.61, because live variables analysis is a "backward" analysis!)

Exercise 12.64: Investigate for some of the abstractions used in analyses presented in the preceding chapters (for example, live variables analysis or reaching definitions analysis) whether or not they are optimal and/or complete.

Exercise 12.65: Consider the instrumented trace semantics for reaching definitions $\langle\!|P|\!\rangle_{RD}$ defined above. Define an abstraction function

$$\alpha : \mathcal{P}(\mathit{Trace}_{RD}) \to \mathcal{P}(\mathit{Trace})$$

that removes the reaching definitions component from the instrumented traces. Show that α induces a Galois connection between $\mathcal{P}(\mathit{Trace}_{RD})$ and $\mathcal{P}(\mathit{Trace})$, and that $\alpha(\langle\!|P|\!\rangle_{RD}) = \langle\!|P|\!\rangle$.

Index of Notation

Bibliography

[All70] Frances E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, July 1970.

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2006.

[AMN17] Esben Sparre Andreasen, Anders Møller, and Benjamin Barslev Nielsen. Systematic approaches for increasing soundness and precision of static analyzers. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis, SOAP@PLDI 2017, Barcelona, Spain, June 18, 2017*, pages 31–36. ACM, 2017.

[And94] Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Copenhagen, 1994.

[BCSS11] Sam Blackshear, Bor-Yuh Evan Chang, Sriram Sankaranarayanan, and Manu Sridharan. The flow-insensitive precision of andersen's analysis in practice. In *Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings*, volume 6887 of *Lecture Notes in Computer Science*, pages 60–76. Springer, 2011.

[BR02] Thomas Ball and Sriram K. Rajamani. The SLAM project: debugging system software via static analysis. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*, pages 1–3. ACM, 2002.

[BS19] Andrew Booker and Andrew Sutherland. Sum of three cubes for 42 finally solved – using real life planetary computer. `https://www.bristol.ac.uk/news/2019/september/sum-of-three-cubes-.html`, 2019.

[CC76] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of programs. In *Proceedings of the 2nd International Symposium on Programming, Paris, France*, pages 106–130. Dunod, 1976.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252. ACM, 1977.

[CC79a] Patrick Cousot and Radhia Cousot. Constructive versions of Tarski's fixed point theorems. *Pacific Journal of Mathematics*, 82(1):43–57, 1979.

[CC79b] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, USA, January 1979*, pages 269–282. ACM, 1979.

[CC92] Patrick Cousot and Radhia Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In *Programming Language Implementation and Logic Programming, 4th International Symposium, PLILP'92, Leuven, Belgium, August 26-28, 1992, Proceedings*, volume 631 of *Lecture Notes in Computer Science*, pages 269–295. Springer, 1992.

[Cha03] Venkatesan T. Chakaravarthy. New results on the computability and complexity of points-to analysis. In *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisiana, USA, January 15-17, 2003*, pages 115–125. ACM, 2003.

[Cha08] Swarat Chaudhuri. Subcubic algorithms for recursive state machines. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 159–169. ACM, 2008.

[Cou77] Patrick Cousot. Asynchronous iterative methods for solving a fixed point system of monotone equations in a complete lattice. Res. rep. R.R. 88, Laboratoire IMAG, Université scientifique et médicale de Grenoble, Grenoble, France, Sep. 1977. 15 p.

[CWZ90] David R. Chase, Mark N. Wegman, and Frank Kenneth Zadeck. Analysis of pointers and structures. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation (PLDI), White Plains, New York, USA, June 20-22, 1990*, pages 296–310. ACM, 1990.

[Dij70] Edsger W. Dijkstra. Notes on structured programming. Technical Report EWD249, Technological University Eindhoven, 1970.

[FFSA98] Manuel Fähndrich, Jeffrey S. Foster, Zhendong Su, and Alexander Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, June 17-19, 1998*, pages 85–96. ACM, 1998.

[FSDF93] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*, pages 237–247. ACM, 1993.

[GF64] Bernard A. Galler and Michael J. Fischer. An improved equivalence algorithm. *Commun. ACM*, 7(5):301–303, 1964.

[GLR15] Roberto Giacobazzi, Francesco Logozzo, and Francesco Ranzato. Analyzing program analyses. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 261–273. ACM, 2015.

[GRS00] Roberto Giacobazzi, Francesco Ranzato, and Francesca Scozzari. Making abstract interpretations complete. *J. ACM*, 47(2):361–416, 2000.

[Hec77] Matthew S. Hecht. *Flow Analysis of Computer Programs*. Elsevier Science Inc., New York, NY, USA, 1977.

[Hen93] Fritz Henglein. Type inference with polymorphic recursion. *ACM Trans. Program. Lang. Syst.*, 15(2):253–289, 1993.

[Hin69] Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, 146:29–60, 1969.

[Hor97] Susan Horwitz. Precise flow-insensitive may-alias analysis is NP-hard. *ACM Trans. Program. Lang. Syst.*, 19(1):1–6, 1997.

[HT01a] Nevin Heintze and Olivier Tardieu. Demand-driven pointer analysis. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001*, pages 24–34. ACM, 2001.

[HT01b] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001*, pages 254–263. ACM, 2001.

[JLB+15] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. A formally-verified C static analyzer. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 247–259. ACM, 2015.

[Kil73] Gary A. Kildall. A unified approach to global program optimization. In *Conference Record of the ACM Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, October 1973*, pages 194–206. ACM, 1973.

[Kle52] Stephen Cole Kleene. *Introduction to Metamathematics*. North-Holland Publ. Comp., Amsterdam, 1952.

[KTU90] Assaf J. Kfoury, Jerzy Tiuryn, and Pawel Urzyczyn. ML typability is DEXPTIME-complete. In *CAAP '90, 15th Colloquium on Trees in Algebra and Programming, Copenhagen, Denmark, May 15-18, 1990, Proceedings*, volume 431 of *Lecture Notes in Computer Science*, pages 206–220. Springer, 1990.

[KTU93] Assaf J. Kfoury, Jerzy Tiuryn, and Pawel Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM Trans. Program. Lang. Syst.*, 15(2):290–311, 1993.

[KU77] John B. Kam and Jeffrey D. Ullman. Monotone data flow analysis frameworks. *Acta Inf.*, 7:305–317, 1977.

[LH03] Ondrej Lhoták and Laurie J. Hendren. Scaling java points-to analysis using SPARK. In *Compiler Construction, 12th International Conference, CC 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, volume 2622 of *Lecture Notes in Computer Science*, pages 153–169. Springer, 2003.

[LSS+15] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z Guyer, Uday P Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: A manifesto. *Communications of the ACM*, 58(2):44–46, 2015.

[Mai90] Harry G. Mairson. Deciding ML typability is complete for deterministic exponential time. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California, USA, January 1990*, pages 382–401. ACM Press, 1990.

[Mid12] Jan Midtgaard. Control-flow analysis of functional programs. *ACM Comput. Surv.*, 44(3):10:1–10:33, 2012.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

[PB09] Fernando Magno Quintão Pereira and Daniel Berlin. Wave propagation and deep propagation for pointer analysis. In *Proceedings of the CGO 2009, The Seventh International Symposium on Code Generation and Optimization, Seattle, Washington, USA, March 22-25, 2009*, pages 126–135. IEEE Computer Society, 2009.

[PKH04] David J. Pearce, Paul H. J. Kelly, and Chris Hankin. Online cycle detection and difference propagation: Applications to pointer analysis. *Softw. Qual. J.*, 12(4):311–337, 2004.

[PKH07] David J. Pearce, Paul H. J. Kelly, and Chris Hankin. Efficient field-sensitive pointer analysis of C. *ACM Trans. Program. Lang. Syst.*, 30(1):4, 2007.

[RC00] Atanas Rountev and Satish Chandra. Off-line variable substitution for scaling points-to analysis. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, British Columbia, Canada, June 18-21, 2000*, pages 47–56. ACM, 2000.

[RHS95] Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, pages 49–61. ACM Press, 1995.

[Ric53] Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[Roo19] Eric Roosendaal. On the 3x + 1 problem. `http://www.ericr.nl/wondrous/`, 2019.

[SCD+13] Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J. Fink, and Eran Yahav. Alias analysis for object-oriented programs. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, volume 7850 of *Lecture Notes in Computer Science*, pages 196–232. Springer, 2013.

[SP81] Micha Sharir and Amir Pnueli. *Two approaches to interprocedural data flow analysis*, chapter 7, pages 189–234. Prentice-Hall, 1981.

[SRH96] Shmuel Sagiv, Thomas W. Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theor. Comput. Sci.*, 167(1&2):131–170, 1996.

[Ste96] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium*

on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996, pages 32–41. ACM, 1996.

[Tar55] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

[Tur37] Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London mathematical society*, 2(1):230–265, 1937.

[Wan87] Mitchell Wand. A simple algorithm and proof for type inference. *Fundamenta Informaticae*, 10(2):115–121, 1987.