Lecture Notes on Static Analysis

Michael I. Schwartzbach
BRICS, Department of Computer Science
University of Aarhus, Denmark
`mis@brics.dk`

Abstract

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

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.

Contents

1 Introduction

There are many interesting questions that can be asked about a given program:

- does the program terminate?
- how large can the heap become during execution?
- what is the possible output?

Other questions concern individual program points in the source code:

- does the variable `x` always have the same value?
- will the value of `x` be read in the future?
- can the pointer `p` be `null`?
- which variables can `p` point to?
- is the variable `x` initialized before it is read?
- is the value of the integer variable `x` always positive?
- what is a lower and upper bound on the value of the integer variable `x`?
- at which program points could `x` be assigned its current value?
- do `p` and `q` point to disjoint structures in the heap?

Rice's theorem is a general result from 1953 that informally can be paraphrased as stating that all interesting questions about the behavior of programs 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. We could exploit this analyzer to also decide the halting problem by using as input the program:

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

Here `x` has a constant value if and only if the `j`'th Turing machine halts on empty input.

This seems like a discouraging result. However, our real focus is not to decide such properties but rather to solve practical problems like making the program run faster or use less space, or finding bugs in the program. The solution is to settle for *approximative* answers that are still precise enough to fuel our applications.

Most often, such approximations are *conservative*, meaning that all errors lean to the same side, which is determined by our intended application.

Consider again the problem of determining if a variable has a constant value. If our intended application is to perform constant propagation, then the analysis may only answer *yes* if the variable really is a constant and must answer *no* if the variable may or may not be a constant. The trivial solution is of course to answer *no* all the time, so we are facing the *engineering* challenge of answering *yes* as often as possible while obtaining a reasonable performance.

A different example is the question: to which variables may the pointer `p` point? If our intended application is to replace `*p` with `x` in order to save a dereference operation, then the analysis may only answer "`&x`" if `p` certainly must point to `x` and must answer "?" if this is false or the answer cannot be determined. If our intended application is instead to determine the maximal size of `*p`, then the analysis must reply with a possibly too large set {`&x,&y,&z,...`} that is guaranteed to contain all targets.

In general, all optimization applications need conservative approximations. If we are given false information, then the optimization is *unsound* and changes the semantics of the program. Conversely, if we are given trivial information, then the optimization fails to do anything.

Approximative answers may also 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. The standard compiler technology, based on type checking, offers little protection from pointer errors. Consider the following small program which performs every kind of error:

```
int main() {
  char *p,*q;
  p = NULL;
  printf("%s",p);
  q = (char *)malloc(100);
  p = q;
  free(q);
  *p = 'x';
  free(p);
  p = (char *)malloc(100);
  p = (char *)malloc(100);
  q = p;
  strcat(p,q);
}
```

The standard tools such as `gcc -Wall` and `lint` detect no errors. If we had even approximative answers to questions about `null` values and pointer targets, then many of the above errors could be caught.

Exercise 1.1: Describe all the errors in the above program.

2 A Tiny Example Language

We use a tiny imperative programming language, called *TIP*, throughout the following sections. It is designed to have a minimal syntax and yet to contain all the constructions that make static analyses interesting and challenging.

Expressions

The basic expressions all denote integer values:

$$\begin{aligned} E &\rightarrow \mathit{intconst} \\ &\rightarrow \mathit{id} \\ &\rightarrow E\ \texttt{+}\ E \mid E\ \texttt{-}\ E \mid E\ \texttt{*}\ E \mid E\ \texttt{/}\ E \mid E\ \texttt{>}\ E \mid E\ \texttt{==}\ E \\ &\rightarrow \texttt{(}\ E\ \texttt{)} \\ &\rightarrow \texttt{input} \end{aligned}$$

The `input` expression reads an integer from the input stream. The comparison operators yield 0 for false and 1 for true. Pointer expressions will be added later.

Statements

The simple statements are familiar:

$$\begin{aligned} S &\rightarrow \mathit{id}\ \texttt{=}\ E\texttt{;} \\ &\rightarrow \texttt{output}\ E\texttt{;} \\ &\rightarrow S\ S \\ &\rightarrow \texttt{if (}E\texttt{)}\ \{\ S\ \} \\ &\rightarrow \texttt{if (}E\texttt{)}\ \{\ S\ \}\ \texttt{else}\ \{\ S\ \} \\ &\rightarrow \texttt{while (}E\texttt{)}\ \{\ S\ \} \\ &\rightarrow \texttt{var}\ \mathit{id}_1\texttt{,}\ldots\texttt{,}\mathit{id}_n\texttt{;} \end{aligned}$$

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. The `var` statement declares a collection of uninitialized variables.

Functions

Functions take any number of arguments and return a single value:

$$F \rightarrow \mathit{id}\ \texttt{(}\ \mathit{id}\texttt{,}\ldots\texttt{,}\mathit{id}\ \texttt{)}\ \{\ \texttt{var}\ \mathit{id}\texttt{,}\ldots\texttt{,}\mathit{id}\texttt{;}\ S\ \texttt{return}\ E\texttt{;}\ \}$$

Function calls are an extra kind of expression:

$$E \rightarrow \mathit{id}\ \texttt{(}\ E\texttt{,}\ldots\texttt{,}E\ \texttt{)}$$

Pointers

Finally, to allow dynamic memory, we introduce pointers into a heap:

$$\begin{aligned} E &\rightarrow \texttt{\&}\mathit{id} \\ &\rightarrow \texttt{malloc} \\ &\rightarrow \texttt{*}E \\ &\rightarrow \texttt{null} \end{aligned}$$

The first expression creates a pointer to a variable, the second expression allocates a new cell in the heap, and the third expression dereferences a pointer value. In order to assign values to heap cells we allow another form of assignment:

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

Note that pointers and integers are distinct values, so pointer arithmetic is not permitted. It is of course limiting that `malloc` only allocates a single heap cell, but this is sufficient to illustrate the challenges that pointers impose.

We also allow function pointers to be denoted by function names. In order to use those, we generalize function calls to:

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

Function pointers serve as a simple model for objects or higher-order functions.

Programs

A program is just a collection of functions:

$P \rightarrow F \ldots F$

The final function is the main 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 make the notationally simplifying assumption that all declared identifiers are unique in a program.

Exercise 2.1: Argue that any program can be normalized so that all declared identifiers are unique.

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) {                      main() {
      var f,q;                        var n;
      if (*p==0) { f=1; }             n = input;
      else {                          return foo(&n,foo);
        q = malloc;                 }
        *q = (*p)-1;
        f=(*p)*((x)(q,x));
      }
      return f;
    }
```

3 Type Analysis

Our programming language is untyped, but of course the various operations are intended to be applied only to certain arguments. Specifically, the following restrictions seem reasonable:

- arithmetic operations and comparisons apply only to integers;
- only integers can be input and output;
- conditions in control structures must be integers;
- only functions can be called; and
- the `*` operator only applies to pointers.

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

Instead of giving up, we resort to a conservative approximation: *typability*. A program is typable if it satisfies a collection of type constraints that is systematically derived from the syntax tree of the given program. This condition implies that the above requirements are guaranteed to hold during execution, but the converse is not true. Thus, our type-checker will be conservative and reject some programs that in fact will not violate any requirements during execution.

Types

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

$\tau \rightarrow$ `int`
$\quad\rightarrow$ `&`τ
$\quad\rightarrow$ `(`τ`,...,`τ`)->`τ

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

Exercise 3.1: Show how regular types can be represented by finite automata so that two types are equal if their automata accept the same language.

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 the unification algorithm.

For each identifier *id* we introduce a type variable $[\![id]\!]$, and for each expression E a type variable $[\![E]\!]$. Here, E refers to a concrete node in the syntax tree—not to the syntax it corresponds to. This makes our notation slightly ambiguous but simpler than a pedantically correct approach. The constraints are systematically defined for each construction in our language:

$intconst$:	$[\![intconst]\!] = \texttt{int}$
E_1 `op` E_2:	$[\![E_1]\!] = [\![E_2]\!] = [\![E_1\ \texttt{op}\ E_2]\!] = \texttt{int}$
E_1`==`E_2:	$[\![E_1]\!] = [\![E_2]\!] \ \wedge\ [\![E_1\texttt{==}E_2]\!] = \texttt{int}$
`input`:	$[\![\texttt{input}]\!] = \texttt{int}$
id `=` E:	$[\![id]\!] = [\![E]\!]$
`output` E:	$[\![E]\!] = \texttt{int}$
`if (`E`)` S:	$[\![E]\!] = \texttt{int}$
`if (`E`)` S_1 `else` S_2:	$[\![E]\!] = \texttt{int}$
`while (`E`)` S:	$[\![E]\!] = \texttt{int}$
id`(`id_1`,...,`id_n`){ ...return` E`; }`:	$[\![id]\!] = \texttt{(}[\![id_1]\!]\texttt{,...,}[\![id_n]\!]\texttt{)->}[\![E]\!]$
id`(`E_1`,...,`E_n`)`:	$[\![id]\!] = \texttt{(}[\![E_1]\!]\texttt{,...,}[\![E_n]\!]\texttt{)->}[\![id\texttt{(}E_1\texttt{,...,}E_n\texttt{)}]\!]$
`(`E`)(`E_1`,...,`E_n`)`:	$[\![E]\!] = \texttt{(}[\![E_1]\!]\texttt{,...,}[\![E_n]\!]\texttt{)->}[\![\texttt{(}E\texttt{)(}E_1\texttt{,...,}E_n\texttt{)}]\!]$
`&`id:	$[\![\texttt{\&}id]\!] = \texttt{\&}[\![id]\!]$
`malloc`:	$[\![\texttt{malloc}]\!] = \texttt{\&}\alpha$
`null`:	$[\![\texttt{null}]\!] = \texttt{\&}\alpha$
`*`E:	$[\![E]\!] = \texttt{\&}[\![\texttt{*}E]\!]$
`*`id`=`E:	$[\![id]\!] = \texttt{\&}[\![E]\!]$

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

Thus, a given program gives rise to a collection of equality constraints on type terms with variables.

Exercise 3.2: Explain each of the above type constraints.

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

Solving Constraints

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

The complicated factorial program generates the following constraints, where duplicates are not shown:

$\llbracket\texttt{foo}\rrbracket = \texttt{(}\llbracket\texttt{p}\rrbracket\texttt{,}\llbracket\texttt{x}\rrbracket\texttt{)->}\llbracket\texttt{f}\rrbracket$
$\llbracket\texttt{*p}\rrbracket = \text{int}$
$\llbracket\texttt{1}\rrbracket = \texttt{int}$
$\llbracket\texttt{p}\rrbracket = \texttt{\&}\llbracket\texttt{*p}\rrbracket$
$\llbracket\texttt{malloc}\rrbracket = \texttt{\&}\alpha$
$\llbracket\texttt{q}\rrbracket = \texttt{\&}\llbracket\texttt{*q}\rrbracket$
$\llbracket\texttt{f}\rrbracket = \llbracket\texttt{(*p)*((x)(q,x))}\rrbracket$
$\llbracket\texttt{(x)(q,x)}\rrbracket = \texttt{int}$
$\llbracket\texttt{input}\rrbracket = \texttt{int}$
$\llbracket\texttt{n}\rrbracket = \llbracket\texttt{input}\rrbracket$
$\llbracket\texttt{foo}\rrbracket = \texttt{(}\llbracket\texttt{\&n}\rrbracket\texttt{,}\llbracket\texttt{foo}\rrbracket\texttt{)->}\llbracket\texttt{foo(\&n,foo)}\rrbracket$
$\llbracket\texttt{*p==0}\rrbracket = \texttt{int}$
$\llbracket\texttt{f}\rrbracket = \llbracket\texttt{1}\rrbracket$
$\llbracket\texttt{0}\rrbracket = \texttt{int}$
$\llbracket\texttt{q}\rrbracket = \llbracket\texttt{malloc}\rrbracket$
$\llbracket\texttt{q}\rrbracket = \texttt{\&}\llbracket\texttt{(*p)-1}\rrbracket$
$\llbracket\texttt{*p}\rrbracket = \texttt{int}$
$\llbracket\texttt{(*p)*((x)(q,x))}\rrbracket = \texttt{int}$
$\llbracket\texttt{x}\rrbracket = \texttt{(}\llbracket\texttt{q}\rrbracket\texttt{,}\llbracket\texttt{x}\rrbracket\texttt{)->}\llbracket\texttt{(x)(q,x)}\rrbracket$
$\llbracket\texttt{main}\rrbracket = \texttt{()->}\llbracket\texttt{foo(\&n,foo)}\rrbracket$
$\llbracket\texttt{\&n}\rrbracket = \texttt{\&}\llbracket\texttt{n}\rrbracket$
$\llbracket\texttt{*p}\rrbracket = \llbracket\texttt{0}\rrbracket$

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

$\llbracket\texttt{p}\rrbracket = \texttt{\&int}$
$\llbracket\texttt{q}\rrbracket = \texttt{\&int}$
$\llbracket\texttt{malloc}\rrbracket = \texttt{\&int}$
$\llbracket\texttt{x}\rrbracket = \phi$
$\llbracket\texttt{foo}\rrbracket = \phi$
$\llbracket\texttt{\&n}\rrbracket = \texttt{\&int}$
$\llbracket\texttt{main}\rrbracket = \texttt{()->int}$

where ϕ is the regular type that corresponds to the infinite unfolding of:

$\phi = \texttt{(\&int,}\phi\texttt{)->int}$

Exercise 3.3: Draw a picture of the unfolding of ϕ.

Since this solution exists, we conclude that our program is type correct. Recursive types are also required for data structures. The example program:

```
var p;
p = malloc;
*p = p;
```

creates the constraints:

$[\![\mathtt{p}]\!] = \mathtt{\&}\alpha$
$[\![\mathtt{p}]\!] = \mathtt{\&}[\![\mathtt{p}]\!]$

which has the solution $[\![\mathtt{p}]\!] = \psi$ where $\psi = \mathtt{\&}\psi$. Some constraints admit infinitely many solutions. For example, the function:

```
poly(x) {
  return *x;
}
```

has type $\mathtt{\&}\alpha\mathtt{->}\alpha$ for any type α, which corresponds to the polymorphic behavior it displays.

Slack and Limitations

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

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

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

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

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

which are clearly unsolvable.

Exercise 3.4: Explain the behavior of this program.

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

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

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

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

The problem is that `*p` denotes a stack cell that has *escaped* from the `baz` function. As we shall see, these problems can instead be handled by more ambitious static analyses.

4 Lattice Theory

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

Lattices

A *partial order* is a mathematical structure: $L = (S, \sqsubseteq)$, where S is a set and $\sqsubseteq$ is a binary relation on S that satisfies the following conditions:

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

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

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

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

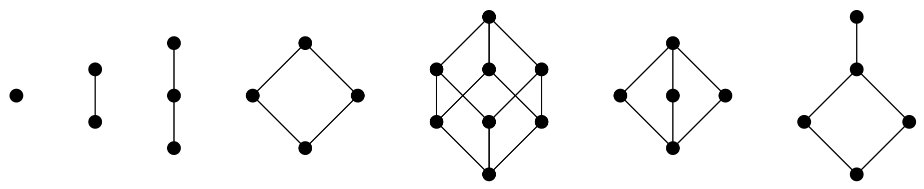
$$\sqcap X \sqsubseteq X \wedge \forall y \in S : y \sqsubseteq X \Rightarrow y \sqsubseteq \sqcap X$$

A *lattice* is a partial order in which $\sqcup X$ and $\sqcap X$ exist for all $X \subseteq S$. Notice that a lattice must have a unique *largest* element $\top$ defined as $\top = \sqcup S$ and a unique *smallest* element $\bot$ defined as $\bot = \sqcap S$.

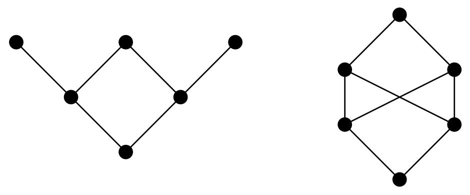
Exercise 4.1: Show that $\sqcup S$ and $\sqcap S$ correspond to $\top$ and $\bot$.

We will often look at *finite* lattices. For those the lattice requirements reduce to observing that $\bot$ and $\top$ exist and that every pair of elements x and y have a least upper bound written $x \sqcup y$ and a greatest lower bound written $x \sqcap y$.

A finite partial order may be illustrated by a 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:

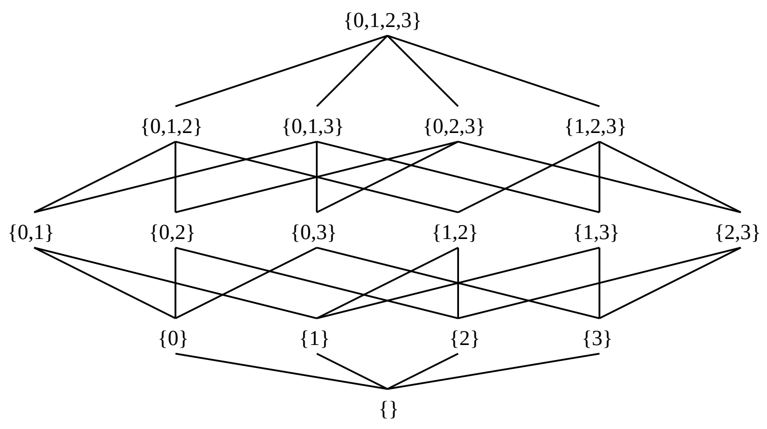


whereas these partial orders are *not* lattices:



Exercise 4.2: Why do these two diagrams not define lattices?

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



The *height* of a lattice is defined to be the length of the longest path from $\bot$ to $\top$. For example, the above powerset lattice has height 4. In general, the lattice $(2^A, \subseteq)$ has height $|A|$.

Fixed-Points

A function $f : L \to L$ is *monotone* when $\forall x, y \in S : x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$. Note that this property does not imply that f is *increasing* $(forallx \in S : x \subseteq f(x))$;

for example, all constant functions are monotone. Viewed as functions $\sqcup$ and $\sqcap$ are monotone in both arguments. Note that the composition of monotone functions is again monotone.

The central result we need is the *fixed-point theorem.* In a lattice L with finite height, every monotone function f has a unique least fixed-point defined as:

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

for which $f(\mathit{fix}(f)) = \mathit{fix}(f)$. 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 $f^i(\bot) \sqsubseteq f^{i+1}(\bot)$. Thus, we have an increasing chain:

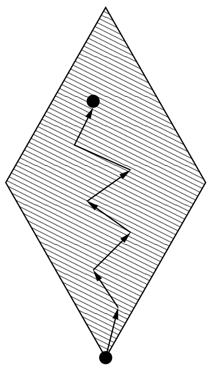
$$\bot \sqsubseteq f(\bot) \sqsubseteq f^2(\bot) \sqsubseteq \ldots$$

Since L is assumed to have finite height, we must for some k have that $f^k(\bot) = f^{k+1}(\bot)$. We define $\mathit{fix}(f) = f^k(\bot)$ and since $f(\mathit{fix}(f)) = f^{k+1}(\bot) = f^k(\bot) = \mathit{fix}(f)$, we know that $\mathit{fix}(f)$ is a fixed-point. Assume now that x is another fixed-point. Since $\bot \sqsubseteq x$ it follows that $f(\bot) \sqsubseteq f(x) = x$, since f is monotone and by induction we get that $\mathit{fix}(f) = f^k(\bot) \sqsubseteq x$. Hence, $\mathit{fix}(f)$ is the least fixed-point. By anti-symmetry, it is also unique.

The time complexity of computing a fixed-point depends on three factors:

- the height of the lattice, since this provides a bound for k;
- the cost of computing f;
- the cost of testing equality.

The computation of a fixed-point can be illustrated as a walk up the lattice starting at $\bot$:



Closure Properties

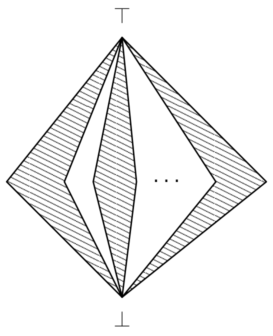
If $L_1, L_2, \ldots, L_n$ are lattices with finite height, then so is the *product*:

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

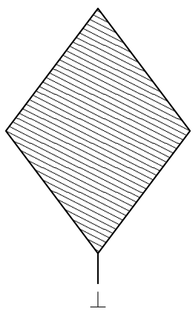
where $\sqsubseteq$ is defined pointwise. Note that $\sqcup$ and $\sqcap$ can be computed pointwise and that $height(L_1 \times \ldots \times L_n) = height(L_1) + \ldots + height(L_n)$. There is also a *sum* operator:

$$L_1 + L_2 + \ldots + L_n = \{(i, x_i) \mid x_i \in L_i \backslash \{\perp, \top\}\} \cup \{\perp, \top\}$$

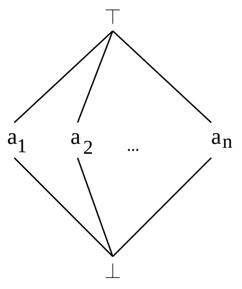
where $\perp$ and $\top$ are as expected and $(i, x) \sqsubseteq (j, y)$ if and only if $i = j$ and $x \sqsubseteq y$. Note that $height(L_1 + \ldots + L_n) = max\{height(L_i)\}$. The sum operator can be illustrated as follows:



If L is a lattice with finite height, then so is $lift(L)$, which can be illustrated by:



and has $height(lift(L)) = height(L) + 1$. If A is a finite set, then $flat(A)$ illustrated by:



is a lattice with height 2. Finally, if A and L are defined as above, then we obtain a *map* lattice with finite height as:

$$A \mapsto L = \{[a_1 \mapsto x_1, \ldots, a_n \mapsto x_n] \mid x_i \in L\}$$

ordered pointwise: $f \sqsubseteq g \Leftrightarrow \forall a_i : f(a_i) \sqsubseteq g(a_i)$. Note that $\mathit{height}(A \mapsto L) = |A| \cdot \mathit{height}(L)$.

Exercise 4.3: Verify the above claims about the heights of the lattices that are constructed.

Equations and Inequations

Fixed-points are interesting because they allow us to solve systems of equations. Let L be a lattice with finite height. An *equation system* is of the form:

$$\begin{aligned} x_1 &= F_1(x_1, \ldots, x_n) \\ x_2 &= F_2(x_1, \ldots, x_n) \\ &\vdots \\ x_n &= F_n(x_1, \ldots, x_n) \end{aligned}$$

where x_i are variables and $F_i : L^n \to L$ is a collection of monotone functions. Every such system has a unique least solution, which is obtained as the least fixed-point of the function $F : L^n \to L^n$ defined by:

$$F(x_1, \ldots, x_n) = (F_1(x_1, \ldots, x_n), \ldots, F_n(x_1, \ldots, x_n))$$

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

$$\begin{aligned} x_1 &\sqsubseteq F_1(x_1, \ldots, x_n) \\ x_2 &\sqsubseteq F_2(x_1, \ldots, x_n) \\ &\vdots \\ x_n &\sqsubseteq F_n(x_1, \ldots, x_n) \end{aligned}$$

by observing that the relation $x \sqsubseteq y$ is equivalent to $x = x \sqcap y$. Thus, solutions are preserved by rewriting the system into:

$$\begin{aligned} x_1 &= x_1 \sqcap F_1(x_1, \ldots, x_n) \\ x_2 &= x_2 \sqcap F_2(x_1, \ldots, x_n) \\ &\vdots \\ x_n &= x_n \sqcap F_n(x_1, \ldots, x_n) \end{aligned}$$

which is just a system of equations with monotone functions as before.

Exercise 4.4: Show that $x \sqsubseteq y$ is equivalent to $x = x \sqcap y$.

5 Control Flow Graphs

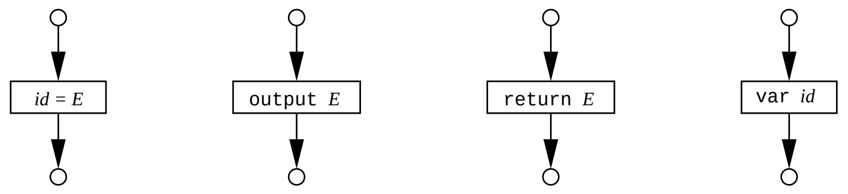
Type analysis started with the syntax tree of a program and defined constraints over variables assigned to nodes. Analyses that work in this manner are *flow* *insensitive*, in the sense that the results remain the same if a statement sequence S_1S_2 is permuted into S_2S_1. Analyses that are *flow sensitive* use a *control flow graph*, which is a different representation of the program source.

For now, we consider only the subset of the TIP language consisting of a single function body without pointers. A control flow graph (CFG) is a directed graph, in which *nodes* correspond to program points and *edges* represent possible flow of control. A CFG always has a single point of entry, denoted *entry*, and a single point of exit, denoted *exit*.

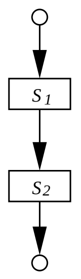
If v is a node in a CFG then *pred*(v) denotes the set of predecessor nodes and *succ*(v) the set of successor nodes.

Control Flow Graphs for Statements

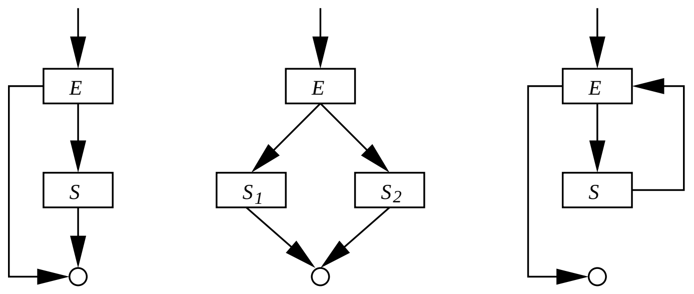
For now, we only consider simple statements, for which CFGs may be constructed in an inductive manner. The CFGs for assignments, `output`, `return` statements, and declarations look as follows:



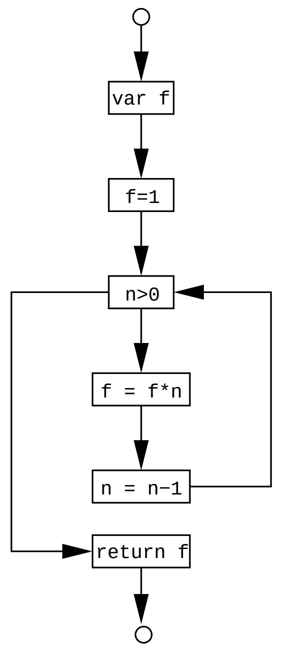
For the sequence S_1 S_2, we eliminate the exit node of S_1 and the entry node of S_2 and glue the statements together:



Similarly, the other control structures are modeled by inductive graph constructions:



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



6 Dataflow Analysis

Classical dataflow analysis, also called the *monotone framework*, starts with a CFG and a lattice L with finite height. The lattice may be fixed for all programs, or it may be *parameterized* with the given program.

To every node v in the CFG, we assign a variable $[\![v]\!]$ ranging over the elements of L. For each construction in the programming language, we then define a *dataflow constraint* that relates the value of the variable of the corresponding node to those of other nodes (typically the neighbors).

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

For a complete CFG, we can systematically extract a collection of constraints over the variables. If all the constraints happen to be equations or inequations with monotone right-hand sides, then we can use the fixed-point algorithm to compute the unique least solution.

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

Fixed-Point Algorithms

If the CFG has nodes $V = \{v_1, v_2, \ldots, v_n\}$, then we work in the lattice L^n. Assuming that node v_i generates the dataflow equation $[\![v_i]\!] = F_i([\![v_1]\!], \ldots, [\![v_n]\!])$, we construct the combined function $F : L^n \to L^n$ as described earlier:

$$F(x_1, \ldots, x_n) = (F_1(x_1, \ldots, x_n), \ldots, F_n(x_1, \ldots, x_n))$$

The naive algorithm is then to proceed as follows:

```
$x = (\perp, \ldots, \perp)$;
do { $t = x$; $x = F(x)$; }
while ($x \neq t$);
```

to compute the fixed-point x. A better algorithm, called *chaotic iteration*, exploits the fact that our lattice has the structure L^n:

```
$x_1 = \perp$; $\ldots x_n = \perp$;
do {
  $t_1 = x_1$; $\ldots t_n = x_n$;
  $x_1 = F_1(x_1, \ldots, x_n)$;
  $\ldots$
  $x_n = F_n(x_1, \ldots, x_n)$;
} while ($x_1 \neq t_1 \lor \ldots \lor x_n \neq t_n$);
```

to compute the fixed-point $(x_1, \ldots, x_n)$.

Exercise 6.1: Why is chaotic iteration better than the naive algorithm?

Both algorithms are, however, clearly wasteful since the information for all nodes is recomputed in every iteration, even though we may know that it cannot have changed. To obtain an even better algorithm, we must study further the structure of the individual constraints.

In the general case, every variable $[\![v_i]\!]$ depends on all other variables. Most often, however, an actual instance of F_i will only read the values of a few other variables. We represent this information as a map:

$$dep : V \to 2^V$$

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. Armed with this information, we can present the *work-list* algorithm:

```
$x_1 = \bot$; $\ldots x_n = \bot$;
$q = [v_1, \ldots, v_n]$;
while ($q \neq []$) {
  assume $q = [v_i, \ldots]$;
  $y = F_i(x_1, \ldots, x_n)$;
  $q = q$.tail();
  if ($y \neq x_i$) {
    for ($v \in dep(v_i)$) $q$.append($v$);
    $x_i = y$;
  }
}
```

to compute the fixed-point $(x_1, \ldots, x_n)$. The worst-case complexity has not changed, but in practice this algorithm saves much time.

Exercise 6.2: Give an invariant that is strong enough to prove the correctness of the work-list algorithm.

Further 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 queue could be changed into a priority queue allowing us to exploit domain-specific knowledge about a particular dataflow problem.

Example: Liveness

A variable is *live* at a program point if its current value may be read during the remaining execution of the program. Clearly undecidable, this property can be approximated by static analysis.

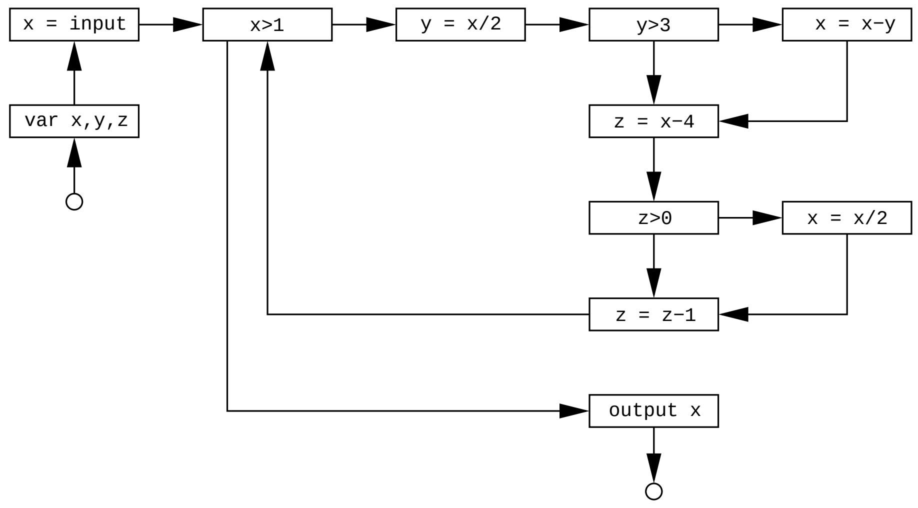
We use a powerset lattice, where the elements are the variables occurring in the given program. For the example program:

```
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 is thus:

$$L = (2^{\{\mathtt{x},\mathtt{y},\mathtt{z}\}}, \subseteq)$$

The corresponding CFG looks as follows:



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

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

For the exit node the constraint is:

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

For conditions and `output` statements, the constraint is:

$$[\![v]\!] = JOIN(v) \cup vars(E)$$

For assignments, the constraint is:

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

For a variable declaration the constraint is:

$$[\![v]\!] = JOIN(v) \setminus \{id_1, \ldots, id_n\}$$

Finally, for all other nodes the constraint is:

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

Here, $vars(E)$ denote the set of variables occurring in E. These constraints clearly have monotone right-hand sides.

Exercise 6.3: Argue that the right-hand sides of constraints define monotone functions.

The intuition is that a variable is live if it is read in the current node, or it is read in some future node unless it is written in the current node. Our example program yields these constraints:

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

whose least solution is:

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

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:

```
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;
```

which saves the cost of one assignment and could result in better register allocation.

We can estimate the worst-case complexity of this analysis. We first observe that if the program has n CFG nodes and k variables, then the lattice has height $k \cdot n$ which bounds the number of iterations we can perform. Each lattice element can be represented as a bitvector of length k. For each iteration we have to perform $O(n)$ intersection, difference, or equality operations which in all takes time $O(kn)$. Thus, the total time complexity is $O(k^2n^2)$.

Example: Available Expressions

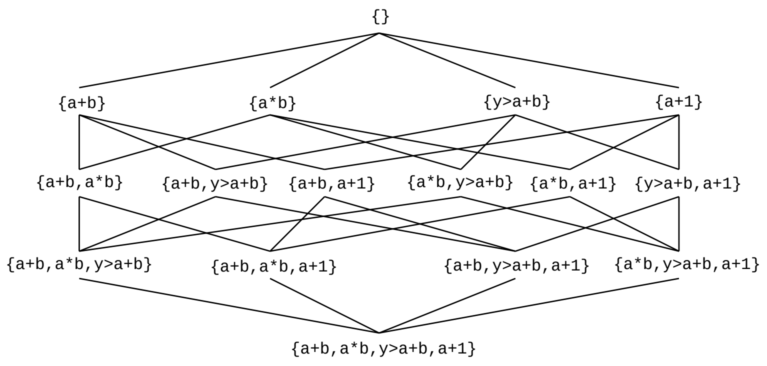
A nontrivial expression in a program is *available* at a program point if its current value has already been computed earlier in the execution. The set of available expressions for all program points can be approximated using a dataflow analysis. The lattice we use has as elements all expressions occurring in the program and is ordered by *reverse* subset inclusion. For a concrete program:

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

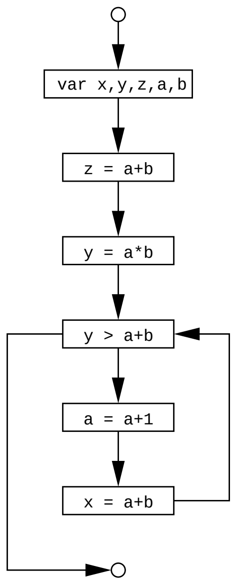
we have 4 different nontrivial expressions, so our lattice is:

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

which looks like:



The largest element of our lattice is $\emptyset$ which corresponds to the trivial information. The flow graph corresponding to the above program is:



For each CFG node v we introduce a constraint variable $[\![v]\!]$ ranging over L. 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 since the computed set may be too small. The dataflow constraints are defined as follows, where we this time define:

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

For the entry node we have the constraint:

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

If v contains a condition E or the statement `output` E, then the constraint is:

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

If v contains an assignment of the form id=E, then the constraint is:

$$[\![v]\!] = (\mathit{JOIN}(v) \cup \mathit{exps}(E)) \downarrow id$$

For all other kinds of nodes, the constraint is just:

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

Here the function $\downarrow id$ removes all expressions that contain a reference to the variable id, and the exps function is defined as:

$exps(intconst) = \emptyset$
$exps(id) = \emptyset$
$exps(input) = \emptyset$
$exps(E_1\,\texttt{op}\,E_2) = \{E_1\,\texttt{op}\,E_2\} \cup exps(E_1) \cup exps(E_2)$

where `op` is any binary operator. The intuition is that an expression is available in v if it is available from all incoming edges or is computed in v, unless its value is destroyed by an assignment statement. Again, the right-hand sides of the constraints are monotone functions. For the example program, we then generate the following concrete constraints:

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

Using the fixed-point algorithm, we obtain the minimal solution:

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

which confirms our assumptions about `a+b`. Observe that the expressions available at the program point *before* a node v can be computed as $JOIN(v)$. 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;
}
```

while being guaranteed of preserving the semantics.

We can estimate the worst-case complexity of this analysis. We first observe that if the program has n CFG nodes and k nontrivial expressions, then the lattice has height $k \cdot n$ which bounds the number of iterations we perform. Each lattice element can be represented as a bitvector of length k. For each iteration we have to perform $O(n)$ intersection, union, or equality operations which in all takes time $O(kn)$. Thus, the total time complexity is $O(k^2n^2)$.

Example: Very Busy Expressions

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

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

The constraint for the exit node is:

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

For conditions and `output` statements we have:

$$[\![v]\!] = JOIN(v) \cup exps(E)$$

For assignments the constraint is:

$$[\![v]\!] = JOIN(v) \downarrow id \ \cup exps(E)$$

For all other nodes we have the constraint:

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

The intuition is that 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. 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 the 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;
```

Example: Reaching Definitions

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 (really 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 becomes:

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

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

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

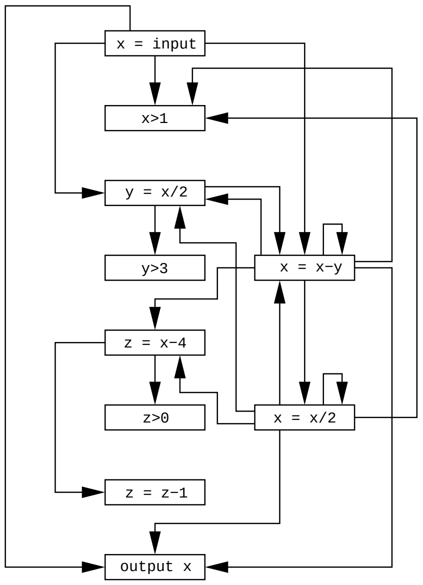
For assignments the constraint is:

$$[\![v]\!] = JOIN(v) \downarrow id \ \ \cup \{v\}$$

and for all other nodes it is simply:

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

This time the $\downarrow id$ function removes all assignments to the variable *id*. This analysis can be used to construct a *def-use graph*, which is like a CFG except that edges go from definitions to possible uses. For the example program, the def-use graph is:



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

Exercise 6.4: Show that the def-use graph is always a subgraph of the transitive closure of the CFG.

Forwards, Backwards, May, and Must

The four classical analyses that we have seen so far can be classified in various ways. They are all just instances of the general monotone framework, but their constraints have a particular structure.

A *forwards* analysis is one that for each program point computes information about the *past* behavior. Examples of this are available expressions and reaching definitions. They can be characterized by the right-hand sides of constraints only depending on *predecessors* of the CFG node. Thus, the analysis starts at the *entry* node and moves forwards in the CFG.

A *backwards* analysis is one that for each program point computes information about the *future* behavior. Examples of this are liveness and very busy expressions. 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 backwards in the CFG.

A *may* analysis is one that describes information that may possibly be true and, thus, computes an *upper* approximation. Examples of this are liveness and reaching definitions. They can be characterized by the right-hand sides of constraints using a *union* operator to combine information.

A *must* analysis is one that describes information that must definitely be true and, thus, computes a *lower* approximation. Examples of this are available expressions and very busy expressions. They can be characterized by the right-hand sides of constraints using an *intersection* operator to combine information.

Thus, our four examples show every possible combination, as illustrated by this diagram:

	Forwards	*Backwards*
May	Reaching Definitions	Liveness
Must	Available Expressions	Very Busy Expressions

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

Example: Initialized Variables

Let us try to define an analysis that ensures that variables are initialized before they are read. This can be solved by computing for every program point the set of variables that is guaranteed to be initialized, thus our lattice is the powerset of variables occurring in the given program. Initialization is a property of the past, so we need a forwards analysis. Also, we need definite information which implies a must analysis. This means that our constraints are phrased in terms of predecessors and intersections. On this basis, they more or less give themselves. For the entry node we have the constraint:

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

for assignments we have the constraint:

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

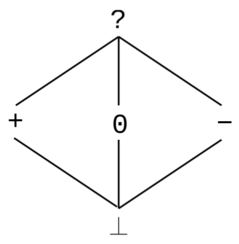
and for all other nodes the constraint:

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

The compiler could now check for every use of a variable that it is contained in the computed set of initialized variables.

Example: Sign Analysis

We now want to determine the sign (+,0,-) of all expressions. So far, every lattice has been the powerset of something, but for this analysis we start with the following tiny lattice *Sign*:



Here, ? denotes that the sign value is not constant and $\perp$ denotes that the value is unknown. The full lattice for our analysis is the map lattice:

$$\mathit{Vars} \mapsto \mathit{Sign}$$

where *Vars* is the set of variables occurring in the given program. For each CFG node v we assign a variable $[\![v]\!]$ that denotes a symbol table giving the sign values for all variables at the program point before the node. The dataflow constraints are more involved this time. For variable declarations we update accordingly:

$$[\![v]\!] = \mathit{JOIN}(v)\ [id_1 \mapsto \mathsf{?}, \ldots, id_n \mapsto \mathsf{?}]$$

For an assignment we use the constraint:

$$[\![v]\!] = \mathit{JOIN}(v)\ [id \mapsto \mathit{eval}(\mathit{JOIN}(v), E)]$$

and for all other nodes the constraint:

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

where:

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

and *eval* performs an abstract evaluation of expressions:

$$\mathit{eval}(\sigma, id) = \sigma(id)$$
$$\mathit{eval}(\sigma, \mathit{intconst}) = \mathit{sign}(\mathit{intconst})$$
$$\mathit{eval}(\sigma, E_1\ \mathsf{op}\ E_2) = \overline{\mathsf{op}}(\mathit{eval}(\sigma, E_1), \mathit{eval}(\sigma, E_2))$$

where σ is the current environment, *sign* gives the sign of an integer constant and $\overline{\mathsf{op}}$ is an abstract evaluation of the given operator, defined by the following collection of tables:

+	$\perp$	0	-	+	?
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
0	$\perp$	0	-	+	?
-	$\perp$	-	-	?	?
+	$\perp$	+	?	+	?
?	$\perp$	?	?	?	?

-	$\perp$	0	-	+	?
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
0	$\perp$	0	+	-	?
-	$\perp$	-	?	-	?
+	$\perp$	+	+	?	?
?	$\perp$	?	?	?	?

*	$\perp$	0	-	+	?
$\perp$	$\perp$	0	$\perp$	$\perp$	$\perp$
0	0	0	0	0	0
-	$\perp$	0	+	-	?
+	$\perp$	0	-	+	?
?	$\perp$	0	?	?	?

/	$\perp$	0	-	+	?
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
0	$\perp$	?	0	0	?
-	$\perp$	?	?	?	?
+	$\perp$	?	?	?	?
?	$\perp$	?	?	?	?

`>`	$\perp$	`0`	`-`	`+`	`?`
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
`0`	$\perp$	`0`	`+`	`0`	`?`
`-`	$\perp$	`0`	`?`	`0`	`?`
`+`	$\perp$	`+`	`+`	`?`	`?`
`?`	$\perp$	`?`	`?`	`?`	`?`

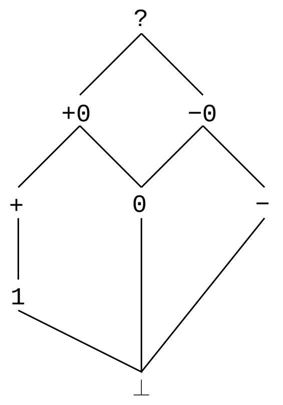
`==`	$\perp$	`0`	`-`	`+`	`?`
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
`0`	$\perp$	`+`	`0`	`0`	`?`
`-`	$\perp$	`0`	`?`	`0`	`?`
`+`	$\perp$	`0`	`0`	`?`	`?`
`?`	$\perp$	`?`	`?`	`?`	`?`

It is not obvious that the right-hand sides of our constraints correspond to monotone functions. However, the $\sqcup$ operator and map updates clearly are, so it all comes down to monotonicity of the abstract operators on the sign lattice. This is best verified by a tedious manual inspection. Notice that for a lattice with n elements, monotonicity of an $n \times n$ table can be verified automatically in time $O(n^3)$.

Exercise 6.5: Describe the $O(n^3)$ algorithm for checking monotonicity of an operator given by an $n \times n$ table.

Exercise 6.6: Check that the above tables indeed define monotone operators on the *Sign* lattice.

We lose some information in the above analysis, since for example the expression `(2>0)==1` is analyzed as `?`, which seems unnecessarily coarse. Also, `+/+` results in `?` rather than `+` since e.g. `1/2` is rounded down to zero. To handle 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:



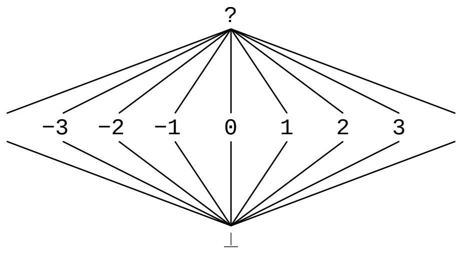
and consequently describe the abstract operators by 8×8 tables.

Exercise 6.7: Define the six operators on the extended *Sign* lattice by means of 8×8 tables. Check that they are properly monotone.

The results of a sign analysis could in theory be used to eliminate division by zero errors by only accepting programs in which denominator expressions have sign `+`, `-`, or `1`. However, the resulting analysis will probably unfairly reject too many programs to be practical.

Example: Constant Propagation

A similar analysis is *constant propagation*, 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 that the basic lattice is replaced by:



and that operators are abstracted in the following manner for e.g. addition:

$$\lambda n \lambda m.\texttt{if}\ (n \neq \texttt{?} \wedge m \neq \texttt{?})\ \{n+m\}\ \texttt{else}\ \{\texttt{?}\}$$

Based on this 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 a dead code elimination, can be reduced to:

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

7 Widening and Narrowing

An *interval analysis* computes for every integer variable a lower and an upper bound for its possible values. The lattice describing a single variable is defined as:

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

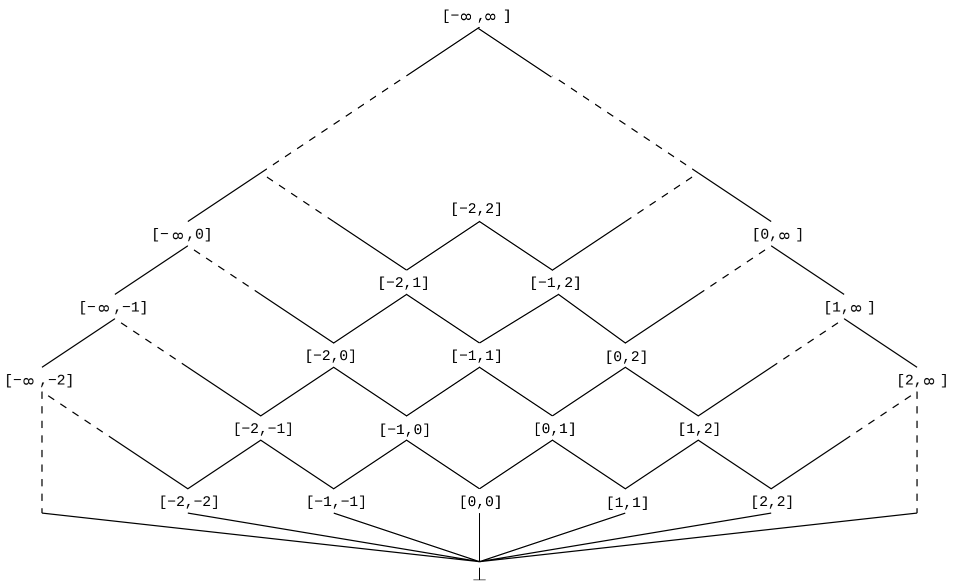
where:

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

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

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

corresponding to inclusion of points. This lattice looks as follows:



It is clear that we do not have a lattice of finite height, since it contains for example the 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 we would ultimately use, namely:

$$L = Vars \mapsto Interval$$

where for the entry node we use the constant function returning the $\top$ element:

$$[\![entry]\!] = \lambda x.[-\infty, \infty]$$

for an assignment the constraint:

$$[\![v]\!] = JOIN(v)\ [id \mapsto eval(JOIN(v), E)]$$

and for all other nodes the constraint:

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

where:

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

and *eval* performs an abstract evaluation of expressions:

$eval(\sigma, id) = \sigma(id)$
$eval(\sigma, intconst) = [intconst, intconst]$
$eval(\sigma, E_1 \,\mathtt{op}\, E_2) = \overline{\mathtt{op}}(eval(\sigma, E_1), eval(\sigma, E_2))$

where the abstract operators all are defined by:

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

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

Exercise 7.1: Argue that these definitions yield monotone operators on the *Interval* lattice.

The lattice has infinite height, so we are unable to use the monotone framework, since the fixed-point algorithm may never terminate. This means that for the lattice L^n the sequence of approximants:

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

need never converge. Instead of giving up, we shall use a technique called *widening* which introduces a function $w : L^n \to L^n$ so that the sequence:

$$(w \circ F)^i(\bot, \ldots, \bot)$$

now converges on a fixed-point that is larger than every $F^i(\bot, \ldots, \bot)$ and thus represents sound information about the program. The widening function w will intuitively *coarsen* the information sufficiently to ensure termination. For our interval analysis, w is defined pointwise down to single intervals. It operates relatively to a fixed finite subset $B \subset N$ that must contain $-\infty$ and ∞. Typically, B could be seeded with all the integer constants occurring in the given program, but other heuristics could also be used. On a single interval we have:

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

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

Exercise 7.2: Show that since w is an increasing monotone function and $w(\mathit{Interval})$ is a finite lattice, the widening technique is guaranteed to work correctly.

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

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

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

Exercise 7.3: Show that $\mathit{fix} \sqsubseteq F^{i+1}(\mathit{fixw}) \sqsubseteq F^i(\mathit{fixw}) \sqsubseteq \mathit{fixw}$.

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

```
y = 0; x = 8;
while (input) {
  x = 7;
  x = x+1;
  y = y+1;
}
```

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

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

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:

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

However, the result for `x` is discouraging. Fortunately, a single application of narrowing refines the result to:

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

which is really the best we could hope for. Correspondingly, further narrowing has no effect. Note that the decreasing sequence:

$$\mathit{fixw} \sqsupseteq F(\mathit{fixw}) \sqsupseteq F^2(\mathit{fixw}) \sqsupseteq F^3(\mathit{fixw}) \ldots$$

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

8 Conditions and Assertions

Until now, we have ignored the values of conditions by simply treating `if`- and `while`-statements as a nondeterministic choice between the two branches. This technique fails to include some information that could potentially be used in a static analysis. Consider for example the following program:

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

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

To exploit the available information, we shall extend the language with two artificial statements: `assert(`E`)` and `refute(`E`)`, where E is a condition from our base language. In the interval analysis, the constraints for these new statement will narrow the intervals for the various variables by exploiting the information that E must be *true* respectively *false*.

The meanings of the conditionals are then 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;
}
refute(x > 0);
```

Constraints for a node v with an `assert` or `refute` statement may trivially be given as:

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

in which case no extra precision is gained. In fact, it requires insight into the specific static analysis to define non-trivial and sound constraints for these constructs.

For the interval analysis, extracting the information carried by general conditions such as E_1 `>` E_2 or E_1 `==` E_2 is complicated and in itself an area of considerable study. For our purposes, we need only consider conditions of the two kinds id `>` E or E `>` id, the first of which for the case of `assert` can be handled by:

$$[\![v]\!] = \mathit{JOIN}(v)[\mathit{id} \mapsto \mathit{gt}(\mathit{JOIN}(v)(\mathit{id}), \mathit{eval}(\mathit{JOIN}(v), E))]$$

where:

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

The cases of `refute` and the dual condition are handled in similar fashions, and all other condtions are given the trivial, but sound identity constraint.

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 8.1: Discuss how more conditions may be given non-trivial constraints for `assert` and `refute`.

9 Interprocedural Analysis

So far, we have only analyzed the body of a single function, which is called an *intraprocedural* analysis. When we consider whole programs containing function calls as well, the analysis is called *interprocedural*. The alternative to this technique is to analyze each function in isolation with maximally pessimistic assumptions about the results of function calls.

Flow Graphs for Programs

We now consider the subset of the TIP language containing functions, but still ignore pointers. The CFG for an entire program is actually quite simple to obtain, since it corresponds to the CFG for a simple program that can be systematically obtained.

First we construct the CFGs for all individual function bodies. All that remains is then to glue them together to reflect function calls properly. This task is made simpler by the fact that we assume all declared identifiers to be unique.

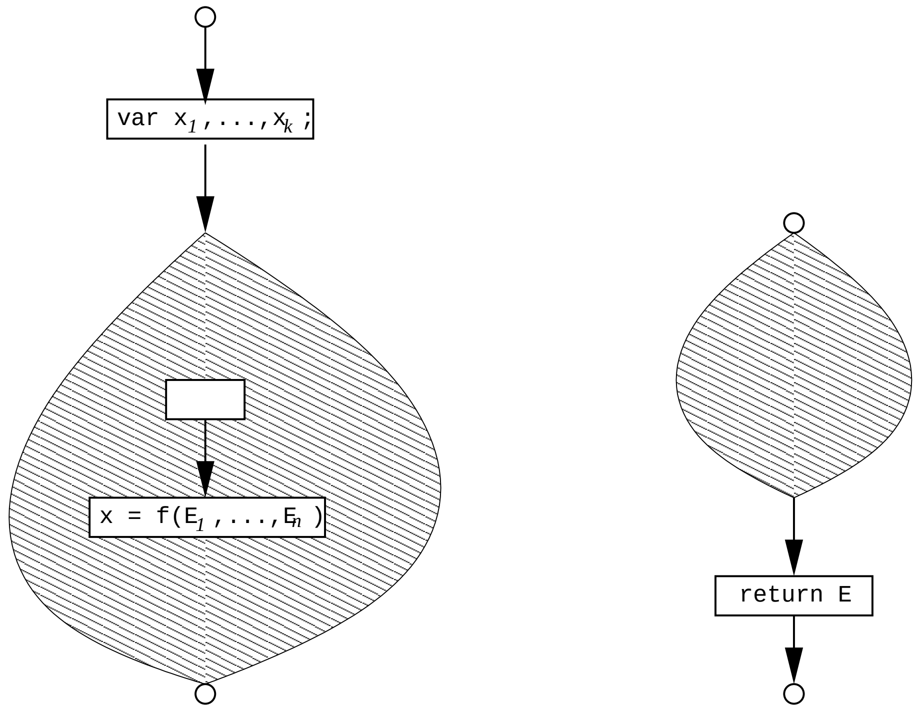
We start by introducing a collection of *shadow* variables. For every function `f` we introduce the variable `ret-f`, which corresponds to its return value. For every call site we introduce a variable `call-i`, where `i` is a unique index, which denotes the value computed by that function call. For every local variable or formal argument named `x` in the calling function and every call site, we introduce a variable `save-i-x` to preserve its value across that function call. Finally, for every formal argument named `x` in the called function and every call site, we introduce a temporary variable `temp-i-x`.

For simplicity we assume that all function calls are performed in connection with assignments:

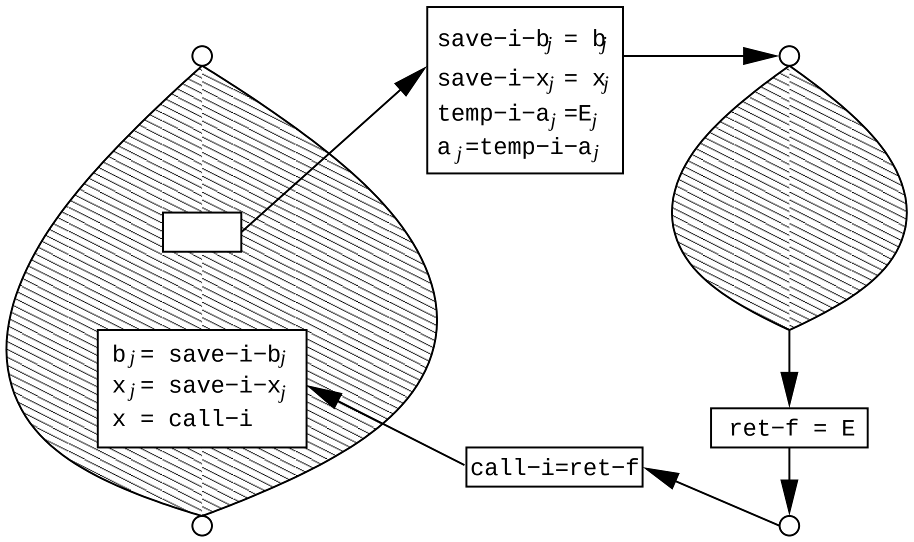
```
x = f(E₁,...,Eₙ);
```

Exercise 9.1: Show how any program can be rewritten to have this form by introducing new temporary variables.

Consider now the CFGs for the calling and the called function:



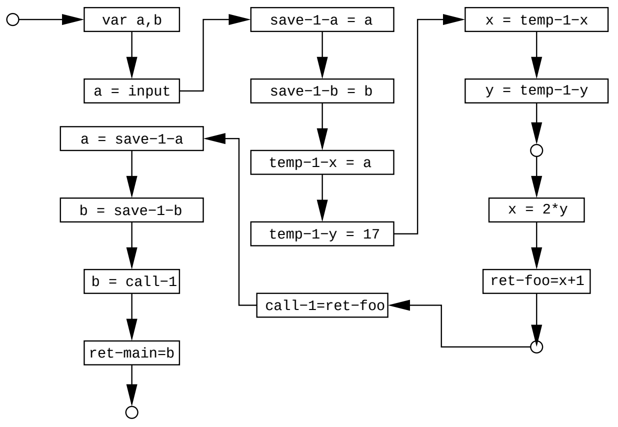
If the formal arguments of the called function are named `a`$_1$,...,`a`$_n$ and those of the calling function are named `b`$_1$,...,`b`$_m$, then the function call transforms the graphs as follows:



which reflects the flow of values during the function call. As a simple example, consider the following program:

```
foo(x,y) {
  x = 2*y;
  return x+1;
}

main() {
  var a,b;
  a = input;
  b = foo(a,17);
  return b;
}
```

The resulting CFG looks as follows:



and can now be analyzed using the standard monotone framework. Note how this construction implies that function arguments are evaluated from left to right. In future examples, the temporary variables will only be used when necessary.

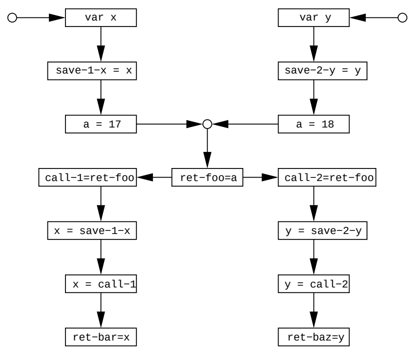
Exercise 9.2: How many edges may the interprocedural CFG contain?

Polyvariance

The interprocedural analysis we have presented so far is called *monovariant*, since each function body is represented only once for every call site. A *polyvariant* analysis will perform context-dependent analysis of function calls. As an example, consider the following program:

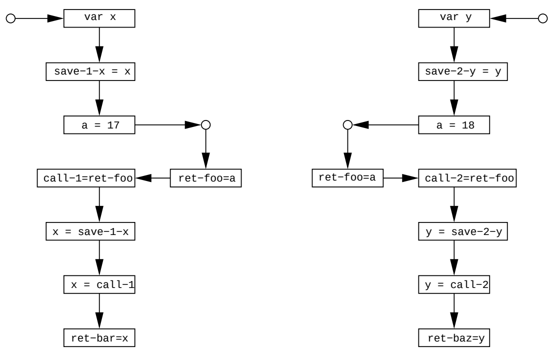
```
foo(a) {
  return a;
}

bar() {                    baz() {
  var x;                     var y;
  x = foo(17);               y = foo(18);
  return x;                  return y;
}                          }
```

which is modeled by this CFG:



If we subsequently perform a constant propagation analysis, then the return values from both `bar` and `baz` are deemed to be non-constant. The problem is that the CFG merges both calls to `foo`.

The analysis can be made polyvariant by creating multiple copies of the CFG for the body of the called function. There are numerous strategies for deciding how many copies to create. The simplest is to create one copy for every call site, which handles the above problem with constant propagation:



If, however, the call to `foo` was wrapped in a further layer of function calls, then nothing would have been gained. Similarly, recursive functions are not benefited much by this technique. The best approach is to employ heuristics that are specific to the intended analysis. It is of course important to ensure that only finitely many copies can be created.

Example: Tree Shaking

An example of an interprocedural analysis is *tree shaking*, where we want to identify those functions that are never called and safely can be removed from the program. This is particularly useful if the program is being compiled together with a large function library.

The analysis takes place on the monovariant interprocedural CFG but is otherwise phrased just like the other analyses we have seen. The lattice is the powerset of function names occurring in the given program, and for every CFG node v we introduce a constraint variable $[\![v]\!]$ denoting the set of functions that could possibly be called in the future. We use the notation $\mathit{entry}(id)$ for the entry node of the function named *id*. For assignments, conditions, and `output` statements the constraint is:

$$[\![v]\!] = \bigcup_{w \in succ(v)} [\![w]\!] \cup \mathit{funcs}(E) \cup \bigcup_{f \in \mathit{funcs}(E)} [\![\mathit{entry}(f)]\!]$$

and for all other nodes just:

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

where *funcs* is defined as:

$\mathit{funcs}(id) = \mathit{funcs}(\mathit{intconst}) = \mathit{funcs}(\texttt{input}) = \emptyset$
$\mathit{funcs}(E_1\, \texttt{op}\, E_2) = \mathit{funcs}(E_1) \cup \mathit{funcs}(E_2)$
$\mathit{funcs}(id\texttt{(}E_1, \ldots, E_n\texttt{)}) = \{id\} \cup \mathit{funcs}(E_1) \cup \ldots \cup \mathit{funcs}(E_n)$

As usual, these constraints can be seen to be monotone. Every function that is not mentioned in the resulting value of $[\![\mathit{entry}(\texttt{main})]\!]$ is guaranteed to be dead.

10 Control Flow Analysis

Interprocedural analysis is fairly straightforward in a language with only first-order functions. If we introduce higher-order functions, objects, or function pointers, then control flow and dataflow suddenly becomes intertwined. The task of *control flow analysis* is to approximate conservatively the control flow graph for such languages.

Closure Analysis for the λ-calculus

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

$E \rightarrow \lambda id.E$
$\quad \rightarrow id$
$\quad \rightarrow E\ E$

and later we shall generalize this technique to the full TIP language. For simplicity we assume that all λ-bound variables are distinct. To construct a CFG for a term in this calculus, we need to compute for every expression E the set of *closures* to which it may evaluate. A closure is in our setting a symbol of the form λid that identifies a concrete λ-abstraction. This problem, called *closure analysis*, can be solved using a variation of the monotone framework. However, since the CFG is not available, the analysis will take place on the syntax tree.

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

$$\{\lambda id\} \subseteq [\![\lambda id.E]\!]$$

(the function may certainly evaluate to itself) and for an application E_1E_2 the *conditional* constraint:

$$\lambda id \in [\![E_1]\!] \Rightarrow [\![E_2]\!] \subseteq [\![id]\!] \ \wedge \ [\![E]\!] \subseteq [\![E_1E_2]\!]$$

for every closure $\lambda id.E$ (the actual argument may flow into the formal argument and the value of the function body is among the possible results of the function call). Note that this is a flow insensitive analysis.

Exercise 10.1: Show how the resulting constraints can be transformed into standard monotone inequations and solved by a fixed-point computation.

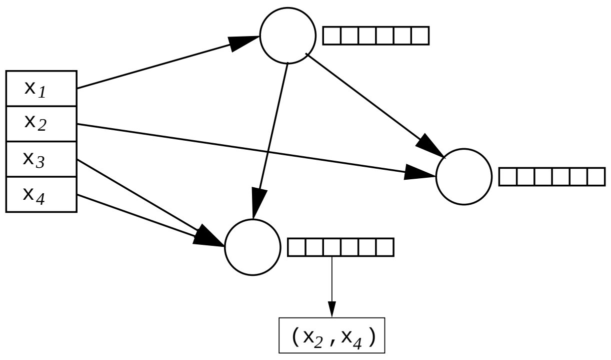
The Cubic Algorithm

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

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

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



Constraints are added one at a time. A constraint of the form $\{t\} \subseteq x$ is handled by looking up the node associated with x and setting the corresponding bit to 1. If its list of pairs was not empty, then an edge between the nodes corresponding to y and z is added for every pair (y, z). A constraint of the form $t \in x \Rightarrow y \subseteq z$ is handled by first testing if the bit corresponding to t in the node corresponding to x has value 1. If this is so, then an edge between the nodes corresponding to y and z is added. Otherwise, the pair (y, z) is added to the list for that bit.

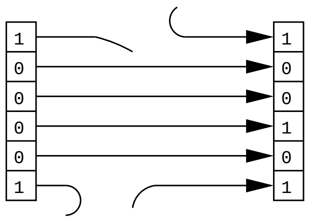
If a newly added edge forms a cycle, then all nodes on that cycle are merged into a single node, which implies that their bitvectors are unioned together and their pair lists are concatenated. The map from variables to nodes is updated accordingly. In any case, to reestablish all inclusion relations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

To analyze this algorithm, we assume that the numbers of tokens and constraints are both $O(n)$. This is clearly the case when analyzing programs, where the numbers of variables, tokens, and constraints all are linear in the size of the program.

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

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

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



Since we have at most n^3 bitwires, the total cost for propagation is $O(n^3)$.

Adding up, the total cost for the algorithm is also $O(n^3)$. The fact that this seems like a lower bound as well is referred to as the *cubic time bottleneck*.

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

Control Flow Graphs for Function Pointers

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

$$E \rightarrow \texttt{(}E\texttt{)(}E_1\texttt{,}\ldots\texttt{,}E_n\texttt{)}$$

we cannot see from the syntax which functions may be called. A coarse but sound CFG could be obtained by assuming that *any* function with the right number of arguments could be called. However, we can do much better by performing a control flow analysis. Note that a function call $id\texttt{(}E_1\texttt{,}\ldots\texttt{,}E_n\texttt{)}$ may be seen as syntactic sugar for the general notation $\texttt{(}id\texttt{)(}E_1\texttt{,}\ldots\texttt{,}E_n\texttt{)}$.

Our lattice is the powerset of the set of tokens containing $\texttt{\&}id$ for every function name *id*, ordered by subset inclusion. For every syntax tree node v we introduce a constraint variable $[\![v]\!]$ denoting the set of functions or function pointers to which v could evaluate. For a constant function name *id* we have the constraint:

$$\{\texttt{\&}id\} \subseteq [\![id]\!]$$

for assignments $id\texttt{=}E$ we have the constraint:

$$[\![E]\!] \subseteq [\![id]\!]$$

and, finally, for computed function calls we have for every definition of a function f with arguments $a_1, \ldots, a_n$ and return expression E' the constraint:

$$\texttt{\&}f \in [\![E]\!] \;\Rightarrow\; [\![E_i]\!] \subseteq [\![a_i]\!] \;\wedge\; [\![E']\!] \subseteq [\![\texttt{(}E\texttt{)(}E_1\texttt{,}\ldots\texttt{,}E_n\texttt{)}]\!]$$

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

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

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

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

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

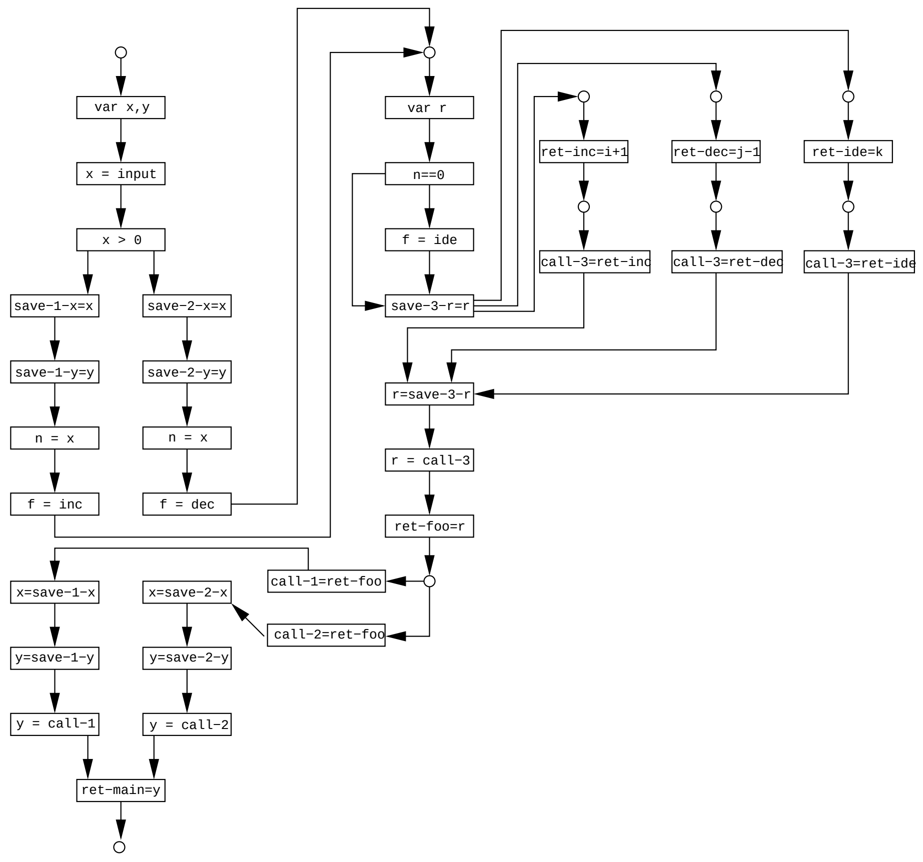
The control flow analysis generates the following constraints:

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

The non-empty values of the least solution are:

$[\![\texttt{inc}]\!] = \{\texttt{\&inc}\}$
$[\![\texttt{dec}]\!] = \{\texttt{\&dec}\}$
$[\![\texttt{ide}]\!] = \{\texttt{\&ide}\}$
$[\![\texttt{f}]\!] = \{\texttt{\&inc}, \texttt{\&dec}, \texttt{\&ide}\}$
$[\![\texttt{foo}]\!] = \{\texttt{\&foo}\}$

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



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

Class Hierarchy Analysis

A language with function pointers or higher-order functions must use this kind of control flow analysis to obtain a reasonably precise CFG. For object-oriented language 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. A final 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 also sufficiently precise.

11 Pointer Analysis

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

Points-To Analysis

The most important information that must be obtained is the set of possible targets of pointers. There are of course infinitely many possible targets during execution, so we must select some finite representatives. The canonical choice is to introduce a target `&`*id* for every variable named *id* and a target `malloc-i`, where `i` is a unique index, for each different allocation site (program point that performs a `malloc` operation). We use *Targets* to denote the set of pointer targets for a given program.

Points-to analysis takes place on the syntax tree, since it will happen before or simultaneously with the control flow analysis. The end result of a points-to analysis is a function pt that for each (pointer) variable p returns the set $pt(p)$ of possible pointer targets to which it may evaluate. We must of course perform a conservative analysis, so these sets will in general be too large.

Given this information, many other facts can be approximated. If we wish to know whether pointer variables p and q *may* be aliases, then a safe answer is obtained by checking whether $pt(p) \cap pt(q)$ is non-empty.

The simplest analysis possible, called *address taken*, is to use all possible targets, except that `&`*id* is only included if this construction occurs in the given program. This only works 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 targets whose types are not equal to that of the pointer variable.

Andersen's Algorithm

One approach to points-to analysis is quite similar to control flow analysis. For each variable named *id* we introduce a set variable $[\![id]\!]$ ranging over the possible pointer targets in the given program.

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

1) id `= malloc`
2) id_1 `= &`id_2

3) id_1 `=` id_2
4) id_1 `= *`id_2
5) `*`id_1 `=` id_2
6) id `= null`

Exercise 11.1: Show how this normalization can be performed systematically by introducing fresh temporary variables.

For each of these pointer manipulations we then generate the following constraints:

$$\begin{array}{rl}
id\ \texttt{= malloc}: & \{\texttt{malloc-i}\} \subseteq [\![id]\!] \\
id_1\ \texttt{= \&}id_2: & \{\texttt{\&}id_2\} \subseteq [\![id_1]\!] \\
id_1\ \texttt{=}\ id_2: & [\![id_2]\!] \subseteq [\![id_1]\!] \\
id_1\ \texttt{= *}id_2: & \texttt{\&}id \in [\![id_2]\!] \Rightarrow [\![id]\!] \subseteq [\![id_1]\!] \\
\texttt{*}id_1\ \texttt{=}\ id_2: & \texttt{\&}id \in [\![id_1]\!] \Rightarrow [\![id_2]\!] \subseteq [\![id]\!]
\end{array}$$

The last two constraints are generated for every variable named *id*, but we need in fact only consider those whose addresses are actually taken in the given program. The `null` assignment is ignored, since it corresponds to the constraint $\emptyset \subseteq [\![id]\!]$. Since these constraints match the requirements of the cubic algorithm, they can be solved in time $O(n^3)$. The resulting points-to function is defined as:

$$pt(p) = [\![p]\!]$$

Consider the following example program:

```
var p,q,x,y,z;
p = malloc;
x = y;
x = z;
*p = z;
p = q;
q = &y;
x = *p;
p = &z;
```

Andersen's algorithm generates these constraints:

$$\begin{array}{l}
\texttt{malloc-1} \subseteq [\![\texttt{p}]\!] \\
[\![\texttt{y}]\!] \subseteq [\![\texttt{x}]\!] \\
[\![\texttt{z}]\!] \subseteq [\![\texttt{x}]\!] \\
\texttt{\&y} \in [\![\texttt{p}]\!] \;\Rightarrow\; [\![\texttt{z}]\!] \subseteq [\![\texttt{y}]\!] \\
\texttt{\&z} \in [\![\texttt{p}]\!] \;\Rightarrow\; [\![\texttt{z}]\!] \subseteq [\![\texttt{z}]\!] \\
[\![\texttt{q}]\!] \subseteq [\![\texttt{p}]\!] \\
\{\texttt{\&y}\} \subseteq [\![\texttt{q}]\!] \\
\texttt{\&y} \in [\![\texttt{p}]\!] \;\Rightarrow\; [\![\texttt{y}]\!] \subseteq [\![\texttt{x}]\!] \\
\texttt{\&z} \in [\![\texttt{p}]\!] \;\Rightarrow\; [\![\texttt{z}]\!] \subseteq [\![\texttt{x}]\!] \\
\{\texttt{\&z}\} \subseteq [\![\texttt{p}]\!]
\end{array}$$

The non-empty values in the least solution are:

$pt(\texttt{p}) = [\![\texttt{p}]\!] = \{\texttt{malloc-1}, \texttt{\&y}, \texttt{\&z}\}$
$pt(\texttt{q}) = [\![\texttt{q}]\!] = \{\texttt{\&y}\}$

which gives a really precise result. Note that while this algorithm is flow insensitive, the directionality of the constraints implies that the dataflow is still modeled with some accuracy.

Steensgaard's Algorithm

A popular alternative performs a coarser analysis essentially by viewing assignments as being bidirectional. This time we use a set consisting of the `malloc-i` tokens and two tokens of the form *id* and `*`*id* for each variable named *id*. We use the same normalized program as before, but this time we generate *equivalence* constraints on tokens:

id `= malloc`:	$*id \sim \texttt{malloc-i}$
id_1 `= &`id_2:	$*id_1 \sim id_2$
id_1 `=` id_2:	$id_1 \sim id_2$
id_1 `= *`id_2:	$id_1 \sim *id_2$
`*`id_1 `=` id_2:	$*id_1 \sim id_2$

The generated constraints induce an equivalence relation on the tokens, which can be computed in almost linear time. The resulting points-to function is defined as:

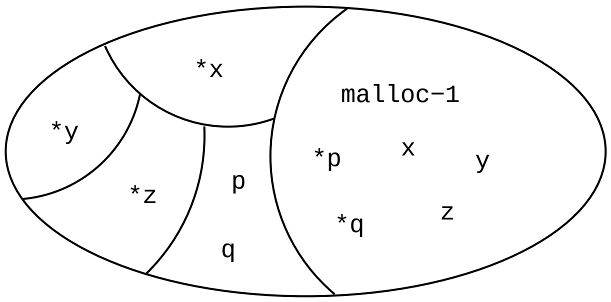
$$pt(p) = \{\texttt{\&}id \mid *p \sim id\} \cup \{\texttt{malloc-i} \mid *p \sim \texttt{malloc-i}\}$$

Again, we might as well restrict ourselves to those instances of `&`*id* that occur in the given program. If we only consider typable programs, then we can further eliminate those targets whose types do not match.

For the previous example program, Steensgaard's algorithm generates the constraints:

$\texttt{*p} \sim \texttt{malloc-1}$	$\texttt{p} \sim \texttt{q}$
$\texttt{x} \sim \texttt{y}$	$\texttt{*q} \sim \texttt{y}$
$\texttt{x} \sim \texttt{z}$	$\texttt{x} \sim \texttt{*p}$
$\texttt{*p} \sim \texttt{z}$	$\texttt{*p} \sim \texttt{z}$

These constraints induce the following equivalence relation:



This in turn implies that:

$pt(\texttt{p}) = pt(\texttt{q}) = \{\texttt{malloc-1}, \texttt{\&x}, \texttt{\&y}, \texttt{\&z}\}$

which is significantly less precise than Andersen's algorithm. Restricting to the addresses that are actually taken, we obtain:

$pt(\texttt{p}) = pt(\texttt{q}) = \{\texttt{malloc-1}, \texttt{\&y}, \texttt{\&z}\}$

which for `p` is as precise as Andersen's algorithm, but still is worse for `q`.

Interprocedural Points-To Analysis

If function pointers are distinguished from other pointers, then we can perform an interprocedural points-to analysis by first computing an interprocedural CFG as described earlier and then running either Andersen's or Steensgaard's algorithm. If, however, function pointers may have indirect references as well then we need to perform the control flow analysis and the points-to analysis simultaneously to resolve for example the function call:

```
(***x)(1,2,3);
```

To express the combined algorithm, we make the syntactic simplification that all function calls are of the form:

id_1 `= (`id_2`)(`a_1`,..., `a_n`);`

where id_i and a_i are variables. Similarly, all return expressions are assumed to be just variables.

Exercise 11.2: Show how to perform these simplifications 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:

$$\{\texttt{\&}f\} \subseteq [\![f]\!]$$

The computed function call generates the constraint:

$$\texttt{\&}f \in [\![id_2]\!] \Rightarrow [\![a_1]\!] \subseteq [\![x_1]\!] \ \wedge \ \ldots \ \wedge \ [\![a_n]\!] \subseteq [\![x_n]\!] \ \wedge \ [\![id]\!] \subseteq [\![id_1]\!]$$

for every occurrence of a function definition:

f`(`x_1`,...,`x_n`) { ...return `id`; }`

This will maintain the precision of the control flow analysis. In contrast, Steensgaard's algorithm would be extended with the constraint:

$$a_1 \sim x_1 \ \wedge \ \ldots \ \wedge \ a_n \sim x_n \ \wedge \ id \sim id_1$$

which results in a considerable loss of precision, since every n-argument function is considered a possible target for the call.

Example: Null Pointer Analysis

We are now also able to define an analysis that detects `null` dereferences. Specifically, we want to ensure that `*p` is only executed when `p` is initialized and does not contain `null`.

As before, we assume that the program is normalized, so that all pointer manipulations are of these kinds:

1) id `= malloc`
2) id_1 `= &`id_2
3) id_1 `=` id_2
4) id_1 `= *`id_2
5) `*`id_1 `=` id_2
6) id `= null`

The basic lattice we use, called *Null*, is:

$$\begin{tikzcd}
\texttt{?} \arrow[d, dash] \\
\texttt{IN} \arrow[d, dash] \\
\texttt{NN} \arrow[d, dash] \\
\bot
\end{tikzcd}$$

where `IN` means *initialized* and `NN` means *not* `null`. We then form the map lattice:

$$\mathit{Vars} \mapsto \mathit{Null}$$

where we recall that *Vars* is the set of variables declared in the given program. For every CFG node v we introduce a constraint variable $[\![v]\!]$ denoting a symbol table giving the status for every variable at that program point. For variable declarations we have the constraint:

$$[\![v]\!] = [id_1 \mapsto \texttt{?}, \ldots, id_n \mapsto \texttt{?}]$$

For the nodes corresponding to the various pointer manipulations we have the constraints:

$$\begin{aligned}
id\ \texttt{= malloc}: \quad & [\![v]\!] = \mathit{JOIN}(v)[id \mapsto \texttt{NN}] \\
id_1\ \texttt{= \&}id_2: \quad & [\![v]\!] = \mathit{JOIN}(v)[id_1 \mapsto \texttt{NN}] \\
id_1\ \texttt{=}\ id_2: \quad & [\![v]\!] = \mathit{JOIN}(v)[id_1 \mapsto \mathit{JOIN}(v)(id_2)] \\
id_1\ \texttt{= *}id_2: \quad & [\![v]\!] = \mathit{right}(\mathit{JOIN}(v), id_1, id_2) \\
\texttt{*}id_1\ \texttt{=}\ id_2: \quad & [\![v]\!] = \mathit{left}(\mathit{JOIN}(v), id_1, id_2) \\
id\ \texttt{= null}: \quad & [\![v]\!] = \mathit{JOIN}(v)[id \mapsto \texttt{IN}]
\end{aligned}$$

and for all other nodes the constraint:

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

where we have defined:

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

$$right(\sigma, x, y) = \sigma[x \mapsto \sigma(y) \sqcup \bigsqcup_{\&p \in pt(y)} \sigma(p)]$$

$$left(\sigma, x, y) = \sigma \underset{\&p \in pt(x)}{} [\; p \mapsto \sigma(p) \sqcup \sigma(y) \;]$$

Note that allocation sites will always be mapped to $\bot$, which reflects that we are not tracking cardinality or connectivity of the heap. After the analysis, the evaluation of `*p` is guaranteed to be safe at program point v if $[\![v]\!](p) = \mathtt{NN}$. The precision of this analysis depends of course on the quality of the underlying points-to analysis.

Exercise 11.3: Explain the above constraints.

Consider the following buggy example program:

```
var p,q,r,n;
p = malloc;
q = &p;
n = null;
*q = n;
*p = r;
```

Andersen's algorithm computes the following points-to sets:

$pt(\mathtt{p}) = \{\mathtt{malloc\text{-}1}\}$
$pt(\mathtt{q}) = \{\mathtt{\&p}\}$
$pt(\mathtt{r}) = \{\}$
$pt(\mathtt{n}) = \{\}$

Based on this information, the `null` pointer analysis generates the following constraints:

$[\![\mathtt{var\ p,q,r,n;}]\!] = [\mathtt{p} \mapsto ?, \mathtt{q} \mapsto ?, \mathtt{r} \mapsto ?, \mathtt{n} \mapsto ?]$
$[\![\mathtt{p=malloc}]\!] = [\![\mathtt{var\ p,q,r,n;}]\!][\mathtt{p} \mapsto \mathtt{NN}]$
$[\![\mathtt{q=\&p}]\!] = [\![\mathtt{p=malloc}]\!][\mathtt{q} \mapsto \mathtt{NN}]$
$[\![\mathtt{n=null}]\!] = [\![\mathtt{q=\&p}]\!][\mathtt{n} \mapsto \mathtt{IN}]$
$[\![\mathtt{*q=n}]\!] = [\![\mathtt{n=null}]\!][\mathtt{p} \mapsto [\![\mathtt{n=null}]\!](\mathtt{p}) \sqcup [\![\mathtt{n=null}]\!](\mathtt{n})]$
$[\![\mathtt{*p=r}]\!] = [\![\mathtt{*q=n}]\!]$

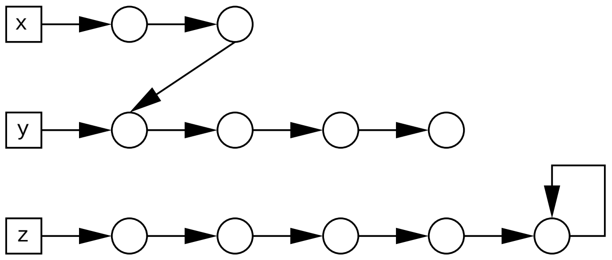
for which the least solution is:

$[\![\mathtt{var\ p,q,r,n;}]\!] = [\mathtt{p} \mapsto ?, \mathtt{q} \mapsto ?, \mathtt{r} \mapsto ?, \mathtt{n} \mapsto ?]$
$[\![\mathtt{p=malloc}]\!] = [\mathtt{p} \mapsto \mathtt{NN}, \mathtt{q} \mapsto ?, \mathtt{r} \mapsto ?, \mathtt{n} \mapsto ?]$
$[\![\mathtt{q=\&p}]\!] = [\mathtt{p} \mapsto \mathtt{NN}, \mathtt{q} \mapsto \mathtt{NN}, \mathtt{r} \mapsto ?, \mathtt{n} \mapsto ?]$
$[\![\mathtt{n=null}]\!] = [\mathtt{p} \mapsto \mathtt{NN}, \mathtt{q} \mapsto \mathtt{NN}, \mathtt{r} \mapsto ?, \mathtt{n} \mapsto \mathtt{IN}]$
$[\![\mathtt{*q=n}]\!] = [\mathtt{p} \mapsto \mathtt{IN}, \mathtt{q} \mapsto \mathtt{NN}, \mathtt{r} \mapsto ?, \mathtt{n} \mapsto \mathtt{IN}]$
$[\![\mathtt{*p=r}]\!] = [\mathtt{p} \mapsto \mathtt{IN}, \mathtt{q} \mapsto \mathtt{NN}, \mathtt{r} \mapsto ?, \mathtt{n} \mapsto \mathtt{IN}]$

By inspecting this information, a compiler could statically detect that when `*p=r` is evaluated, the variable `p` may contain `null` and the variable `r` may be uninitialized.

Example: Shape Analysis

So far, we have viewed the heap as an amorphous structure and only answered questions about stack based variables. The heap can be analyzed in more detail using *shape analysis*. Note that we can produce interesting heaps, even though the `malloc` operation only allocates a single heap cell. An example of a non-trivial 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.

Shape analysis requires a more ambitious lattice of *shape graphs*, which are directed graphs in which the nodes are the pointer targets for the given program. Shape graphs are ordered by inclusion of their sets of edges. Thus, $\perp$ is the graph without edges and $\top$ is the completely connected graph. The pointer targets serve as an abstraction of all the cells that could possibly be created during execution, and the existence of an edge implies that the store *may* contain a reference between two cells that are represented by the source and target nodes. Formally, our lattice is then:

$$2^{\mathit{Targets} \times \mathit{Targets}}$$

ordered by the usual subset inclusion. For every CFG node v we introduce a constraint variable $[\![v]\!]$ denoting a shape graph that describes all possible stores after that program point. For the nodes corresponding to the various pointer manipulations we have the constraints:

$$\begin{array}{rl}
id\ \texttt{= malloc}: & [\![v]\!] = \mathit{JOIN}(v)\!\downarrow\! id\ \cup\ \{(\texttt{\&}id, \texttt{malloc-i})\} \\
id_1\ \texttt{=}\ \texttt{\&}id_2: & [\![v]\!] = \mathit{JOIN}(v)\!\downarrow\! id_1\ \cup\ \{(\texttt{\&}id_1, \texttt{\&}id_2)\} \\
id_1\ \texttt{=}\ id_2: & [\![v]\!] = \mathit{assign}(\mathit{JOIN}(v), id_1, id_2) \\
id_1\ \texttt{=}\ \texttt{*}id_2: & [\![v]\!] = \mathit{right}(\mathit{JOIN}(v), id_1, id_2) \\
\texttt{*}id_1\ \texttt{=}\ id_2: & [\![v]\!] = \mathit{left}(\mathit{JOIN}(v), id_1, id_2) \\
id\ \texttt{= null}: & [\![v]\!] = \mathit{JOIN}(v)\!\downarrow\! id
\end{array}$$

and for all other nodes the constraint:

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

where we have defined:

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

$$\sigma \downarrow x = \{(s,t) \in \sigma \mid s \neq \&x\}$$

$$assign(\sigma,x,y) = \sigma \downarrow x \;\cup \bigcup_{(\&y,t) \in \sigma} \{(\&x,t)\}$$

$$right(\sigma,x,y) = \sigma \downarrow x \;\cup \bigcup_{(\&y,s),(s,t) \in \sigma} \{(\&x,t)\}$$

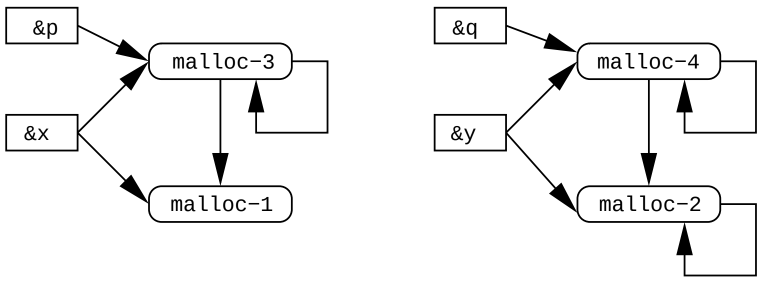
$$left(\sigma,x,y) = \begin{cases} \sigma & \{s \mid (\&x,s) \in \sigma\} = \emptyset \\ \bigcup_{(\&x,s) \in \sigma} \sigma \downarrow s & \{s \mid (\&x,s) \in \sigma\} \neq \emptyset \wedge \{t \mid (\&y,t) \in \sigma\} = \emptyset \\ \bigcup_{(\&x,s),(\&y,t) \in \sigma} \sigma \downarrow s \;\cup\; \{(s,t)\} & \textit{otherwise} \end{cases}$$

Exercise 11.4: Explain the above constraints.

Consider now the following program:

```
var x,y,n,p,q;
x = malloc; y = malloc;
*x = null; *y = y;
n = input;
while (n>0) {
  p = malloc; q = malloc;
  *p = x; *q = y;
  x = p; y = q;
  n = n-1;
}
```

After the loop, the analysis produces the following shape graph:



From this result we can safely conclude that `x` and `y` will always be disjoint.

Note that our shape analysis also computes a flow sensitive points-to map that for each program point v is defined by:

$$pt(p) = \{t \mid (\texttt{\&}p, t) \in [\![v]\!]\}$$

This analysis is more precise than Andersen's algorithm, but clearly also more expensive to perform. As an example, consider the program:

```
x = &y;
x = &z;
```

After these statements, Andersen's algorithm would predict that $pt(\texttt{x}) = \{\texttt{\&y}, \texttt{\&z}\}$ whereas the shape analysis computes $pt(\texttt{x}) = \{\texttt{\&z}\}$ for the final program point. This flow sensitive points-to information could be used to boost the `null` pointer analysis. However, an initial flow insensitive points-to analysis would still be required to construct a CFG for programs using function pointers. Conversely, if we have another points-to analysis, then it may be used to boost the precision of the shape analysis by restricting the targets considered in the *left* and *right* functions.

Example: Better Shape Analysis

The above shape analysis allows us to conclude that `x` and `y` will always be disjoint. However, the shape graphs we compute are unable to answer other interesting questions. For example, we cannot conclude that `malloc-2` nodes always contain a self-loop whereas `malloc-4` nodes never appear on cycles. To make such distinctions, we need a more detailed lattice. As an example, we could maintain information about cyclicity in the graph. This would change our lattice into:

$$2^{Targets \times Targets} \times 2^{Targets \times Targets}$$

where for an element (X, Y) we have that X denotes the possible edges and $Y \subseteq X$ denotes those edges that could possibly be part of a cycle in the graph (thus, Y is used to remember part of the history of the heap).

With this more detailed lattice we now get the obligation of correspondingly updating the constraints. Of course, we could trivially always consider the second component to contain all edges, but to obtain useful results we need to do better. An assignment *id* `= malloc` can never create a cycle, so the corresponding constraint may be updated as follows (where we assume that $JOIN(v) = (X, Y)$):

$$id\ \texttt{= malloc:}\ [\![v]\!] = (X \downarrow id \cup \{(\texttt{\&}id, \texttt{malloc-i})\}, Y \downarrow id)$$

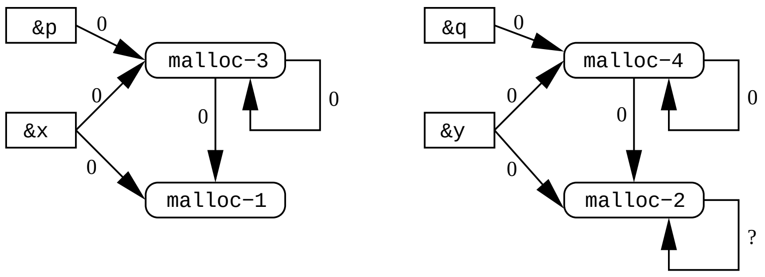
An assignment id_1 `=` `&`id_2 cannot create a cycle if the current shape graph does not contain a path from `&`id_2 to `&`id_1 (since reachability in the shape graph is conservative). Consequently, the constraint may be updated as follows:

id_1 `=` $\&id_2$: $[\![v]\!] = (X \downarrow id_1 \cup \{(\&id_1, \&id_2)\}, Y \downarrow id_1 \cup \mathit{reach}(X \downarrow id_1, \&id_2, \&id_1))$

where $\mathit{reach}(\sigma, s, t)$ returns (t, s) if s can reach t through zero or more edges in σ, and $\emptyset$ otherwise.

Exercise 11.5: Show how the remaining constraints are similarly updated.

With this more detailed analysis, we then compute the following shape graph:



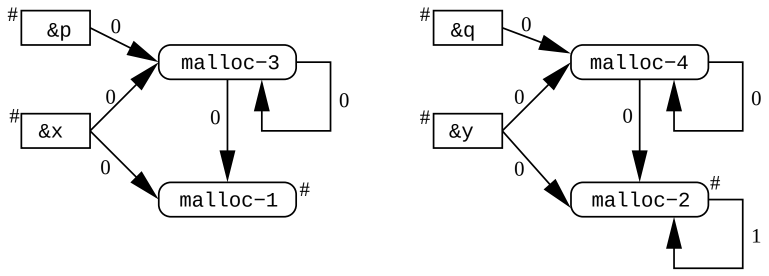
where the edge label 0 means that the edge is definitely not on a cycle and ? means that the answer is unknown. Now, we are able to conclude that the `malloc-4` nodes never appear on cycles. We are allowed to think that the `malloc-2` nodes appear on cycles, but we still do not have enough information to conclude that a self-loop is formed. To substantiate this conclusion, we need to further refine our lattice to keep track of edges that definitely are parts of cycles and to keep track of whether more than one node of a given target can ever be allocated. This requires further refinements of the lattice as follows:

$$2^{\mathit{Targets} \times \mathit{Targets}} \times 3^{\mathit{Targets} \times \mathit{Targets}} \times 2^{\mathit{Targets}}$$

We here use the notation 3^A to denote the lattice $A \mapsto \mathit{3Val}$, where $\mathit{3Val}$ is the lattice of 3-valued Booleans:

$$\begin{tikzcd}[column sep=small] & ? \arrow[dl, no head] \arrow[dr, no head] & \\ 0 \arrow[dr, no head] & & 1 \arrow[dl, no head] \\ & \bot & \end{tikzcd}$$

where 0 means definitely false, 1 means definitely true, and ? means unknown. In a lattice element (X, Y, Z), X denotes the shape graph edges, Y denotes (may or must) knowledge about cyclicity of edges, and Z denotes those targets of which only a single instance has ever been allocated. We can now define even more complicated constraints that maintain this information, and the analysis will result in the following information (where `#` denotes that a target is uniquely allocated):



Finally, we are able to conclude that there is only a single `malloc-2` node and that it has a self-loop.

Analyses like the above can be performed in a less ad-hoc manner, using a framework known as *parametric* shape analysis. Here, the targets are characterized by a number of unary *instrumentation* predicates that are chosen to provide the information necessary for the analysis. Examples of such unary predicates are:

- does this node have two or more incoming pointers?
- is this node reachable from the variable `x`?
- is this node on a cycle?

but the relevant ones depend on the questions for which we seek answers. Our simple shape graphs had a single node for each pointer target. In the parametric framework, we polyvariantly have a copy for each possible 3-valued interpretation of the predicates. Thus, the nodes of a shape graph corresponds to:

$$3^{Targets} \times 3^{Targets} \times \ldots \times 3^{Targets}$$

with one copy for each predicate. The shape graph itself must then describe the connectivity between these nodes, and again we use 3-valued logic to describe edges as definitely present, definitely absent, or possibly present. Thus, our final lattice becomes:

$$3^{(3^{Targets} \times 3^{Targets} \times \ldots \times 3^{Targets})^2}$$

or (amusingly):

$$3^{((3^{Targets})^k)^2}$$

if we have k predicates. To complete the parametric shape analysis, we must the specify the constraints. If we have included many predicates, then we have a correspondingly heavy burden of maintaining these with useful precision. This is generally something of a puzzle to get right, as indicated even by our simple example above. But the technique is powerful and may be used to verify e.g. that a procedure for inserting into a red-black search tree respects the red-black invariant.

Example: Escape Analysis

We earlier lamented the *escaping stack cell* error displayed by the program:

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

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

which was beyond the scope of the type system. Having performed the simple shape analysis, we can easily perform an *escape analysis* to catch such errors. We just need to check that the possible pointer targets for return expressions in the shape graph cannot reach arguments or variables defined in the function itself, since all other pointer targets must then necessarily reside in earlier frames on the invocation stack.

12 Conclusion

We have seen the basic tools that are required to perform static analysis of programs. Real-life applications invariably gravitate back to the techniques that we have covered, though many variations and extensions are usually required.

Two major areas have not been covered at all. The *quality* of an analysis can only be measured relatively to a suite of intended applications. It is rare that competing analyses can be formally compared, so much work in this area is concerned with performing experiments to establish the precision and efficiency of proposed analyses. The *correctness* of an analysis requires a formal semantics of the underlying programming language. Completely formal proofs of correctness of analyses are exceedingly laborious and remain mostly academic exercises. Even so, it is often possible to provide convincing correctness arguments.

Acknowledgement

Thanks to Aske Simon Christensen and Anders Møller for comments and ideas.