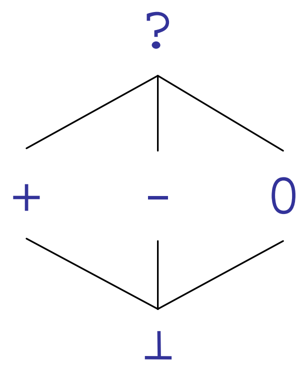
Dataflow Analysis
Widening and Narrowing
Path Sensitivity
Interprocedural Analysis

Static Analysis 2009

Michael I. Schwartzbach
Computer Science, University of Aarhus

Sign Analysis

- Determine the sign (+,-,0) of all expressions
- The *Sign* lattice:



- The full lattice is the map lattice: *Vars* → *Sign*
  - where *Vars* is the set of variables in the program

Sign Constraints

- The variable [[v]] denotes a map that gives the sign value for all variables at the program point *after* v

- For variable declarations:

  $[[v]] = [id_1 \rightarrow ?, ..., id_n \rightarrow ?]$

- For assignments:

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

- For all other nodes:

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

Evaluating Signs

- The *eval* function is an *abstract evaluation*:
  - $eval(\sigma, id) = \sigma(id)$
  - $eval(\sigma, intconst) = sign(intconst)$
  - $eval(\sigma, E_1 \mathrm{op} E_2) = \overline{\mathrm{op}}(eval(\sigma, E_1), eval(\sigma, E_2))$
- The *sign* function gives the sign of an integer
- The $\overline{\mathrm{op}}$ function is an abstract evaluation of the given operator

Abstract Operators

+	⊥	0	-	+	?
⊥	⊥	⊥	⊥	⊥	⊥
0	⊥	0	-	+	?
-	⊥	-	-	?	?
+	⊥	+	?	+	?
?	⊥	?	?	?	?

-	⊥	0	-	+	?
⊥	⊥	⊥	⊥	⊥	⊥
0	⊥	0	+	-	?
-	⊥	-	?	-	?
+	⊥	+	+	?	?
?	⊥	?	?	?	?

*	⊥	0	-	+	?
⊥	⊥	0	⊥	⊥	⊥
0	0	0	0	0	0
-	⊥	0	+	-	?
+	⊥	0	-	+	?
?	⊥	0	?	?	?

/	⊥	0	-	+	?
⊥	⊥	⊥	⊥	⊥	⊥
0	⊥	?	0	0	?
-	⊥	?	?	?	?
+	⊥	?	?	?	?
?	⊥	?	?	?	?

>	⊥	0	-	+	?
⊥	⊥	⊥	⊥	⊥	⊥
0	⊥	0	+	0	?
-	⊥	0	?	0	?
+	⊥	+	+	?	?
?	⊥	?	?	?	?

==	⊥	0	-	+	?
⊥	⊥	⊥	⊥	⊥	⊥
0	⊥	+	0	0	?
-	⊥	0	?	0	?
+	⊥	0	0	?	?
?	⊥	?	?	?	?

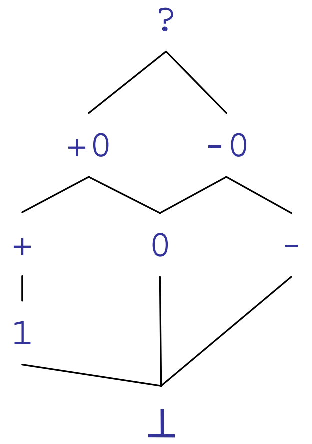
Monotonicity

- The $\sqcup$ operator and map updates are monotone
- Compositions preserve monotonicity
- Are the abstract operators monotone?

- This is verified by a tedious manual inspection
- Or better, run an $O(n^3)$ algorithm for an $n \times n$ table:
  - $\forall x,y,x' \in L$: $x \sqsubseteq x' \Rightarrow x\ \overline{\mathrm{op}}\ y \sqsubseteq x'\ \overline{\mathrm{op}}\ y$
  - $\forall x,y,y' \in L$: $y \sqsubseteq y' \Rightarrow x\ \overline{\mathrm{op}}\ y \sqsubseteq x\ \overline{\mathrm{op}}\ y'$

Increasing Precision

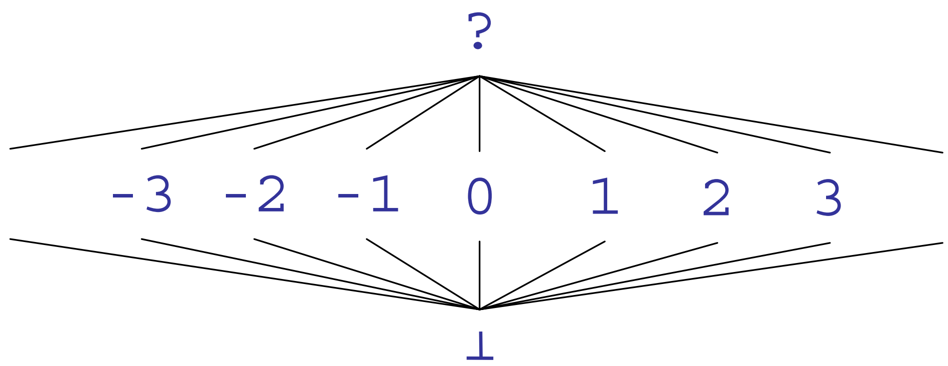
- Some loss of information:
  - `(2>0)==1` is analyzed as `?`
  - `+/+` is analyzed as `?`, since e.g. ½ is rounded down
- Use a richer lattice for better precision:



- Abstract operators are now 8×8 tables

Constant Propagation

- Determine variables with a constant value
- Similar to sign analysis, with basic lattice:



- Abstract operator for addition:

$$\overline{+}(n,m) = \texttt{if}\ (n \neq ? \wedge m \neq ?)\ \{\ n+m\ \}\ \texttt{else}\ \{\ ?\ \}$$

Constant Folding

- Exploiting constant propagation:

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

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

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

Interval Analysis

- Compute upper and lower bounds for integers
- Lattice of intervals:

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

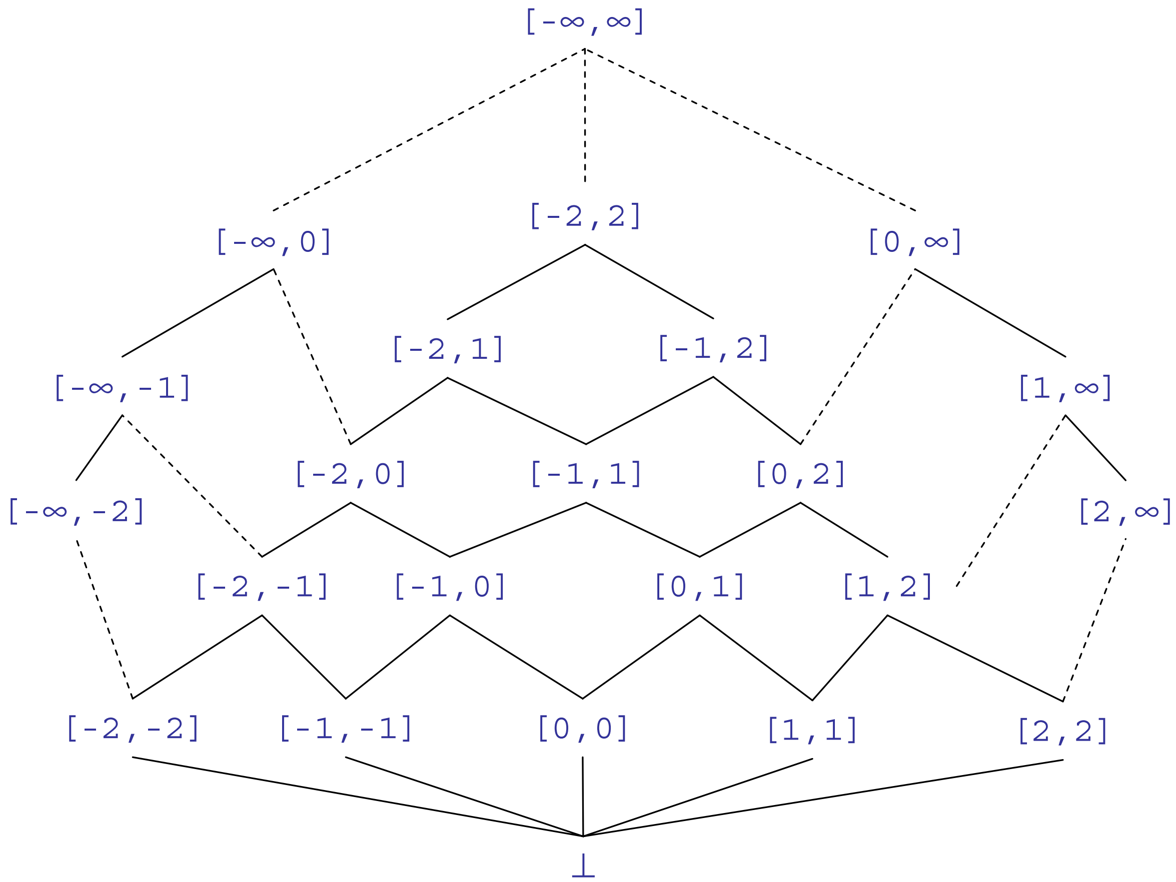
  where:

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

  and intervals are ordered by inclusion:

  $$[l_1, h_1] \sqsubseteq [l_2, h_2] \text{ iff } l_2 \le l_1 \wedge h_1 \le h_2$$

The Interval Lattice



Interval Analysis Lattice

- The total lattice for a program point is:

  L = *Vars* $\rightarrow$ *Interval*

  that provides bounds for each (integer) variable

- This lattice has *infinite height*, since the chain:

  `[0,0]` $\sqsubseteq$ `[0,1]` $\sqsubseteq$ `[0,2]` $\sqsubseteq$ `[0,3]` $\sqsubseteq$ `[0,4]` ...

  occurs in *Interval*

Interval Constraints

- For the *entry* node:

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

- For assignments:

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

- For all other nodes:

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

Evaluating Intervals

- The *eval* function is an *abstract evaluation*:
  - $eval(\sigma, id) = \sigma(id)$
  - $eval(\sigma, intconst) = [intconst, intconst]$
  - $eval(\sigma, E_1 \text{ op } E_2) = \overline{\text{op}}(eval(\sigma, E_1), eval(\sigma, E_2))$
- Abstract arithmetic operators:
  - $\overline{\text{op}}([l_1, h_1], [l_2, h_2]) =$

$$[\min_{x \in [l_1, h_1],\, y \in [l_2, h_2]} x \text{ op } y,\ \max_{x \in [l_1, h_1],\, y \in [l_2, h_2]} x \text{ op } y]$$

- Abstract comparison operators:
  - $\overline{\text{op}}([l_1, h_1], [l_2, h_2]) = [0, 1]$

Fixed-Point Problems

- The lattice has infinite height, so the fixed-point algorithm does not work

- In L^n the sequence of approximants:

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

  need never converge

Widening

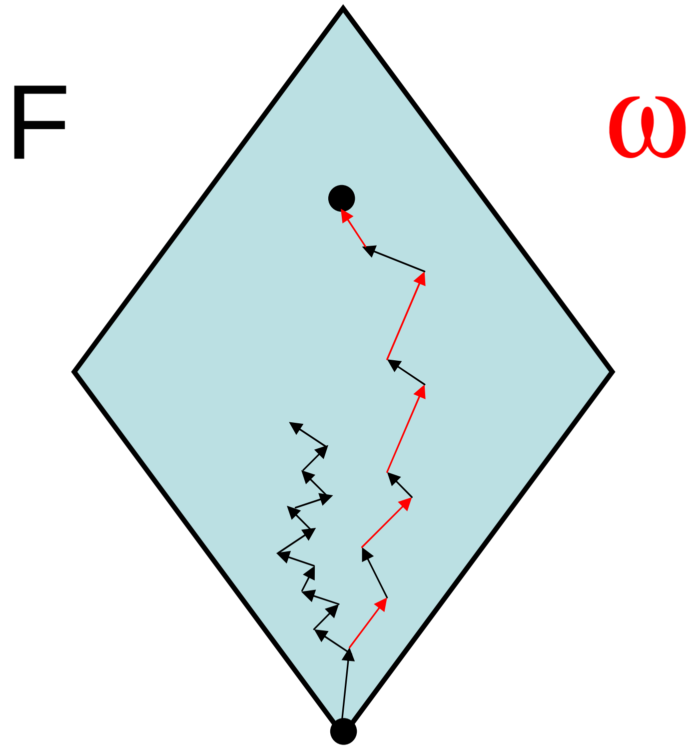
- Introduce a *widening* function $\omega: L^n \rightarrow L^n$ so that:

  $$(\omega \circ F)^i(\perp, \perp, \ldots, \perp)$$

  converges on a fixed-point that is larger than all of the approximants $F^i(\perp, \perp, \ldots, \perp)$

- The function ω coarsens the information

Turbo Charging



Widening for Intervals

- The function ω is defined pointwise
- Parameterized with a fixed finite subset $B \subset N$
  - must contain $-\infty$ and ∞
  - typically seeded with all integer constants occurring in the given program
- On single intervals:

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

- Finds the nearest enclosing allowed interval

Correctness of Widening

- Widening works when:
  - ω is an *increasing* and *monotone* function
  - $\omega(L)$ is a *finite* lattice
- $F^i(\bot, \bot, ..., \bot) \sqsubseteq (\omega \circ F)^i(\bot, \bot, ..., \bot)$
  since F is monotone and ω is increasing
- $\omega \circ F$ is a monotone function $\omega(L) \to \omega(L)$
  so the fixed-point exists

Narrowing

- Widening shoots over the target
- *Narrowing* may improve the result by applying F
- Define:

  $fix = \bigsqcup F^i(\bot, \bot, ..., \bot)$ $fix\omega = \bigsqcup (\omega \circ F)^i(\bot, \bot, ..., \bot)$

  then $fix \sqsubseteq fix\omega$
- But we also have that:

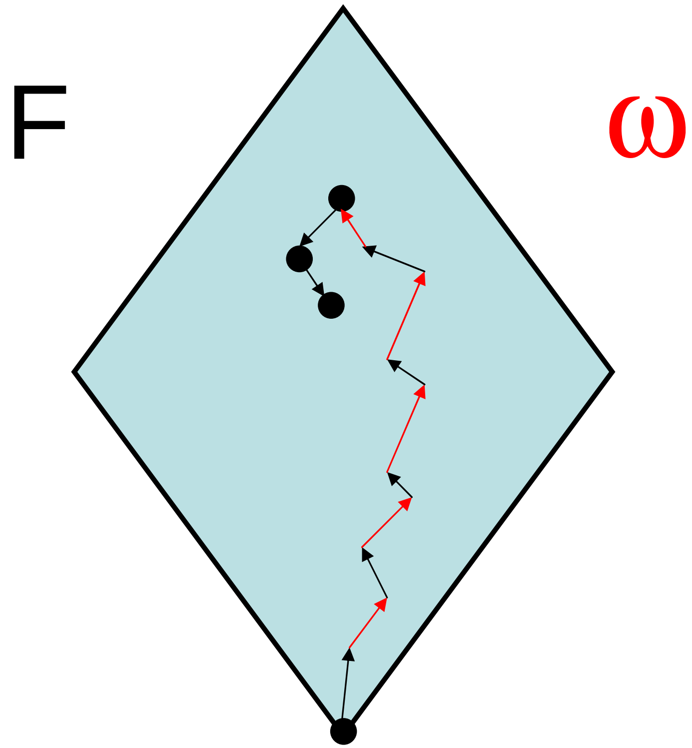
  $fix \sqsubseteq F(fix\omega) \sqsubseteq fix\omega$

  so applying F again may improve the result
- This can be iterated arbitrarily many times

Correctness of Narrowing

- $F(\mathit{fix}\omega) \sqsubseteq \omega(F(\mathit{fix}\omega)) = (\omega \circ F)(\mathit{fix}\omega) = \mathit{fix}\omega$
  - by induction and monotonicity of F we also have:
    $F^{i+1}(\mathit{fix}\omega) \sqsubseteq F^{i}(\mathit{fix}\omega) \sqsubseteq \mathit{fix}\omega$
- $\mathit{fix} = \bigsqcup F^{i}(\bot, \bot, \ldots, \bot) = \bigsqcup F^{i+1}(\bot, \bot, \ldots, \bot)$
  $\sqsubseteq F(\bigsqcup F^{i}(\bot, \bot, \ldots, \bot)) = F(\mathit{fix}) \sqsubseteq F(\mathit{fix}\omega)$
  - by induction we also have:
    $\mathit{fix} \sqsubseteq F^{i}(\mathit{fix}\omega)$

Backing Up



Divergence in Action

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

$[x \rightarrow \bot, y \rightarrow \bot]$
$[x \rightarrow [8,8], y \rightarrow [0,1]]$
$[x \rightarrow [8,8], y \rightarrow [0,2]]$
$[x \rightarrow [8,8], y \rightarrow [0,3]]$
…

Widening in Action

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

$[x \to \bot, y \to \bot]$
$[x \to [7,\infty], y \to [0,1]]$
$[x \to [7,\infty], y \to [0,7]]$
$[x \to [7,\infty], y \to [0,\infty]]$

$B = \{-\infty, 0, 1, 7, \infty\}$

Narrowing in Action

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

$[x \rightarrow \bot, y \rightarrow \bot]$
$[x \rightarrow [7,\infty], y \rightarrow [0,1]]$
$[x \rightarrow [7,\infty], y \rightarrow [0,7]]$
$[x \rightarrow [7,\infty], y \rightarrow [0,\infty]]$

$[x \rightarrow [8,8], y \rightarrow [0,\infty]]$

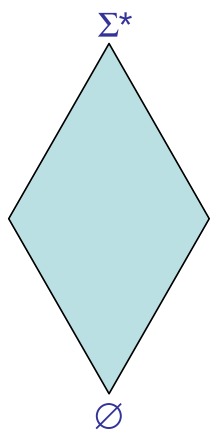
$B = \{-\infty, 0, 1, 7, \infty\}$

Widening Functions

- A simple generic widening function:

$$\omega(x) = \begin{cases} x & \text{if } x \text{ is small enough} \\ \top & \text{otherwise} \end{cases}$$

- A difficult widening function (regular languages):



$\{a\} \subseteq \{a,ab\} \subseteq \{a,ab,abb\} \subseteq \dots \xrightarrow{\omega} \{ab^*\}$

This is essentially machine learning...

Information in Conditions

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

- The interval analysis (with widening) concludes:

  x = [-∞,∞], y = [0,∞], z = [-∞,∞]

Modeling Conditions

- Add two artifical statements

- The statement `assert(`*E*`)` models that *E* is *true* in the current program state
- It causes a runtime error otherwise

- The statement `refute(`*E*`)` models that *E* is *false* in the current program state
- It causes a runtime error otherwise

Encoding Conditions

```
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);
```

Preserves semantics since `assert` and `refute` are guarded by conditions

Constraints for Assert and Refute

- A trivial but sound constraint is:

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

- A non-trivial constraint for `assert(`*id*`>`*E*`)`:

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

  where

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

- Similar constraints are defined for the dual cases
- More tricky to define for all conditions...

Exploiting Conditions

```
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);
```

- The interval analysis now concludes:

  $x = [-\infty, 0]$, $y = [0, 17]$, $z = [0, \infty]$

Branch Correlations

- With assert and refute we have a simple form of *path sensitivity*

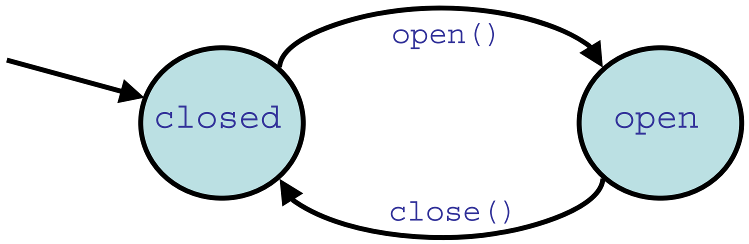
- But it is insufficient to handle *correlation* of branches in program:

```
if (17 > x) { ... }
...
if (17 > x) { ... }
...
```

Open and Closed Files

- Built-in functions `open()` and `close()` on a file

- Requirements:
  - never `close` a closed file
  - never `open` an open file



- We want a static analysis to check this...

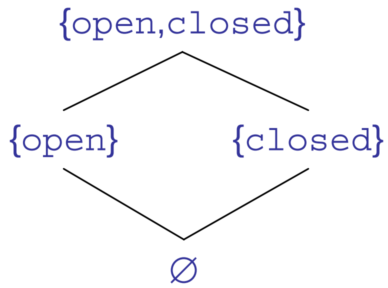
A Tricky Example

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

The Naive Analysis (1/2)

- The lattice models the status of the file:

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



- For every CFG node, v, we have a constraint variable [[v]] denoting the status *after* v

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

The Naive Analysis (2/2)

- Constraints for interesting statements:

  [[*entry*]] = {`closed`}

  [[`open()`]] = {`open`}

  [[`close()`]] = {`closed`}

- For all other CFG nodes:

  [[v]] = *JOIN*(v)

- Before the `close()` statement the analysis concludes that the file is {`open`,`closed`}

Context Awareness

- We need to keep track of the `flag` variable
- Our second attempt is the lattice:

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

- Additionally, we add `assert(...)` and `refute(...)` to keep track of conditionals

- Even so, we now only now that the file is {`open`,`closed`} and that flag is {`flag=0`,`flag≠0`}

Relational Analysis

- We need an analysis that keeps track of *relations* between variables
- This requires that we maintain *multiple* abstract states per program point, one for each *context*
- For the file example we need the lattice:

$$L = C \rightarrow 2^{\{\texttt{open},\texttt{closed}\}}$$

where $C = \{\texttt{flag=0},\texttt{flag}\neq\texttt{0}\}$ is the set of contexts

Enhanced Program

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

Relational Constraints (1/2)

- For the file statements:

  $[\![entry]\!] = \lambda c.\{\texttt{closed}\}$

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

  $[\![\texttt{closed()}]\!] = \lambda c.\{\texttt{closed}\}$

infeasible

- For `flag` assignments:

  $[\![\texttt{flag = 0}]\!] = [\texttt{flag=0} \to \bigcup_{c \in C} JOIN(v)(c),\ \texttt{flag} \neq \texttt{0} \to \varnothing]$

  $[\![\texttt{flag = } n]\!] = [\texttt{flag} \neq \texttt{0} \to \bigcup_{c \in C} JOIN(v)(c),\ \texttt{flag=0} \to \varnothing]$

  $[\![\texttt{flag = } E]\!] = \lambda d.\bigcup_{c \in C} JOIN(v)(c)$

Relational Constraints (2/2)

- For assert and refute statements:

$$[\![\texttt{assert(flag)}]\!] = [\texttt{flag}\neq 0 \rightarrow \mathit{JOIN}(v)(\texttt{flag}\neq 0),\ \texttt{flag}=0 \rightarrow \emptyset]$$

$$[\![\texttt{refute(flag)}]\!] = [\texttt{flag}=0 \rightarrow \mathit{JOIN}(v)(\texttt{flag}=0),\ \texttt{flag}\neq 0 \rightarrow \emptyset]$$

- For all other CFG nodes:

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

Generated Constraints

$[\![\textit{entry}]\!] = \lambda c.\{\texttt{closed}\}$

$[\![\texttt{condition}]\!] = [\![\textit{entry}]\!]$

$[\![\texttt{assert(condition)}]\!] = [\![\texttt{condition}]\!]$

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

$[\![\texttt{flag = 1}]\!] = [\texttt{flag}\neq 0 \rightarrow \cup_{c\in C}[\![\texttt{open()}]\!](c), \texttt{flag=0} \rightarrow \varnothing]$

$[\![\texttt{refute(condition)}]\!] = \texttt{condition}$

$[\![\texttt{flag = 0}]\!] = [\texttt{flag=0} \rightarrow \cup_{c\in C}[\![\texttt{refute(condition)}]\!](c), \texttt{flag}\neq 0 \rightarrow \varnothing]$

$[\![\ldots]\!] = \lambda c.([\![\texttt{flag = 1}]\!](c) \cup [\![\texttt{flag = 0}]\!](c))$

$[\![\texttt{flag}]\!] = [\![\ldots]\!]$

$[\![\texttt{assert(flag)}]\!] = [\![\texttt{flag}\neq 0 \rightarrow [\![\texttt{flag}]\!](\texttt{flag}\neq 0), \texttt{flag=0} \rightarrow \varnothing]$

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

$[\![\texttt{refute(flag)}]\!] = [\texttt{flag=0} \rightarrow [\![\texttt{flag}]\!](\texttt{flag=0}), \texttt{flag}\neq 0 \rightarrow \varnothing]$

$[\![\textit{exit}]\!] = \lambda c.([\![\texttt{close()}]\!](c) \cup [\![\ldots]\!](c))$

Minimal Solution

	`flag = 0`	`flag ≠ 0`
[[*entry*]]	{closed}	{closed}
[[condition]]	{closed}	{closed}
[[assert(condition)]]	{closed}	{closed}
[[open()]]	{open}	{open}
[[flag = 1]]	∅	{open}
[[refute(condition)]]	{closed}	{closed}
[[flag = 0]]	{closed}	∅
[[...]]	{closed}	{open}
[[flag]]	{closed}	{open}
[[assert(flag)]]	∅	{open}
[[close()]]	{closed}	{closed}
[[refute(flag)]]	{closed}	∅
[[*exit*]]	{closed}	{open}

- We know the file is `open` before `close()`

Challenges

- The static analysis designer must choose C
  - often as combinations of predicates from conditionals
  - *iterative refinement* gradually adds predicates
- Exponential blow-up:
  - for *k* predicates, we have 2^k different contexts
  - redundancy often cuts this down
- Reasoning about `assert` and `refute`:
  - how to update the lattice elements sufficiently precisely
  - possibly involves theorem proving

Improvements

- Run auxiliary analyses first, for example:
  - constant propagation
  - sign analysis

  will help in handling `flag` assignments

- Dead code propagation, change:

  [[`open()`]] = λc.{`open`}

  into the still sound but more precise:

  [[`open()`]] = λc.if *JOIN*(v)(c)=∅ then ∅ else {`open`}

Interprocedural Analysis

- Analyzing the body of a single function:
  - *intra*procedural analysis
- Analyzing the whole program with function calls:
  - *inter*procedural analysis

- The alternative is to:
  - analyze each function in isolation
  - be maximally pessimistic about results of function calls

CFG for Whole Programs

- Construct a CFG for each function
- Then glue them together to reflect function calls

- Assume that all function calls are of the form:

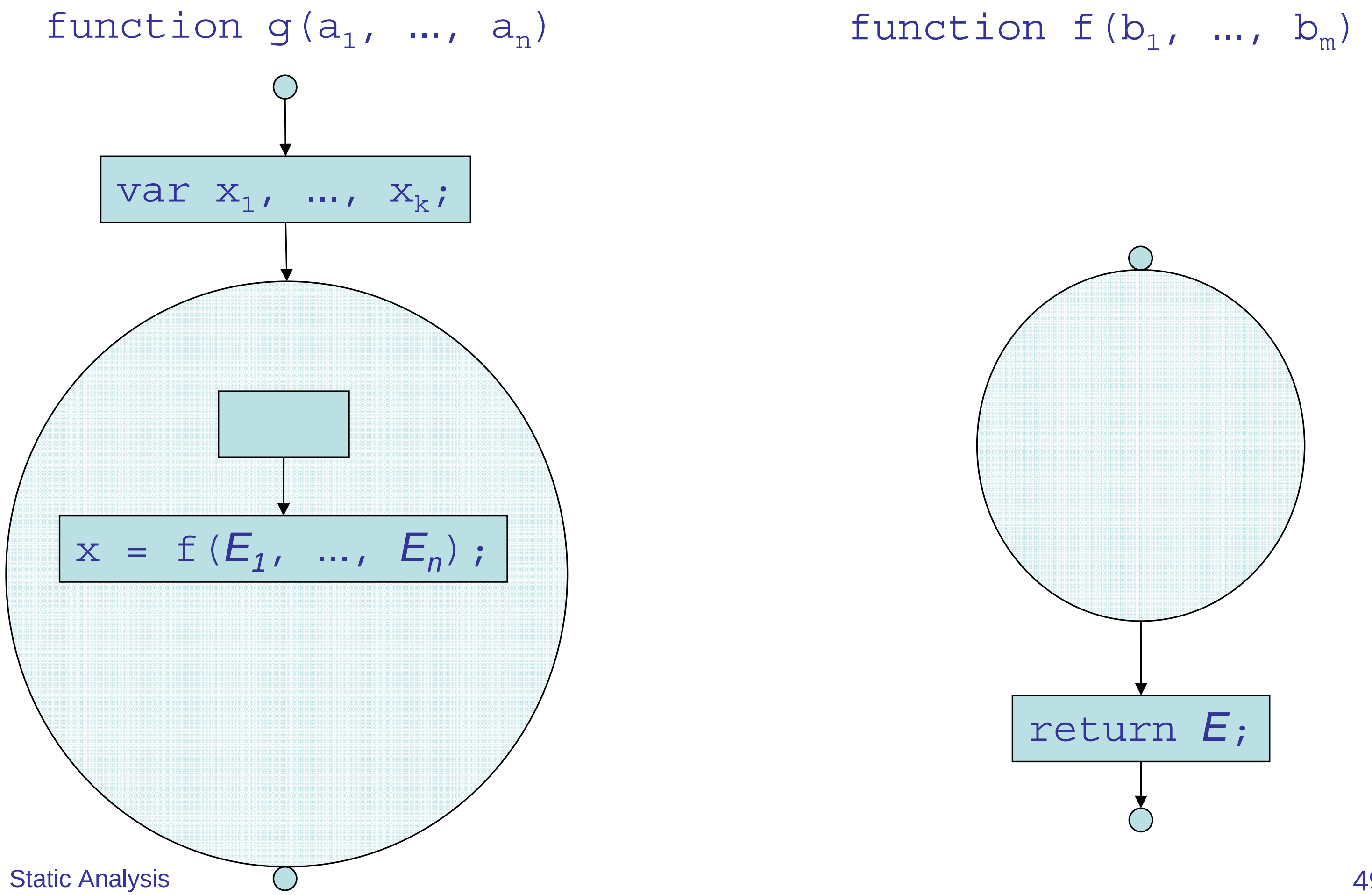
$$id \texttt{ = f(} E_1, \ldots, E_n \texttt{);}$$

- This can always be obtained by rewriting

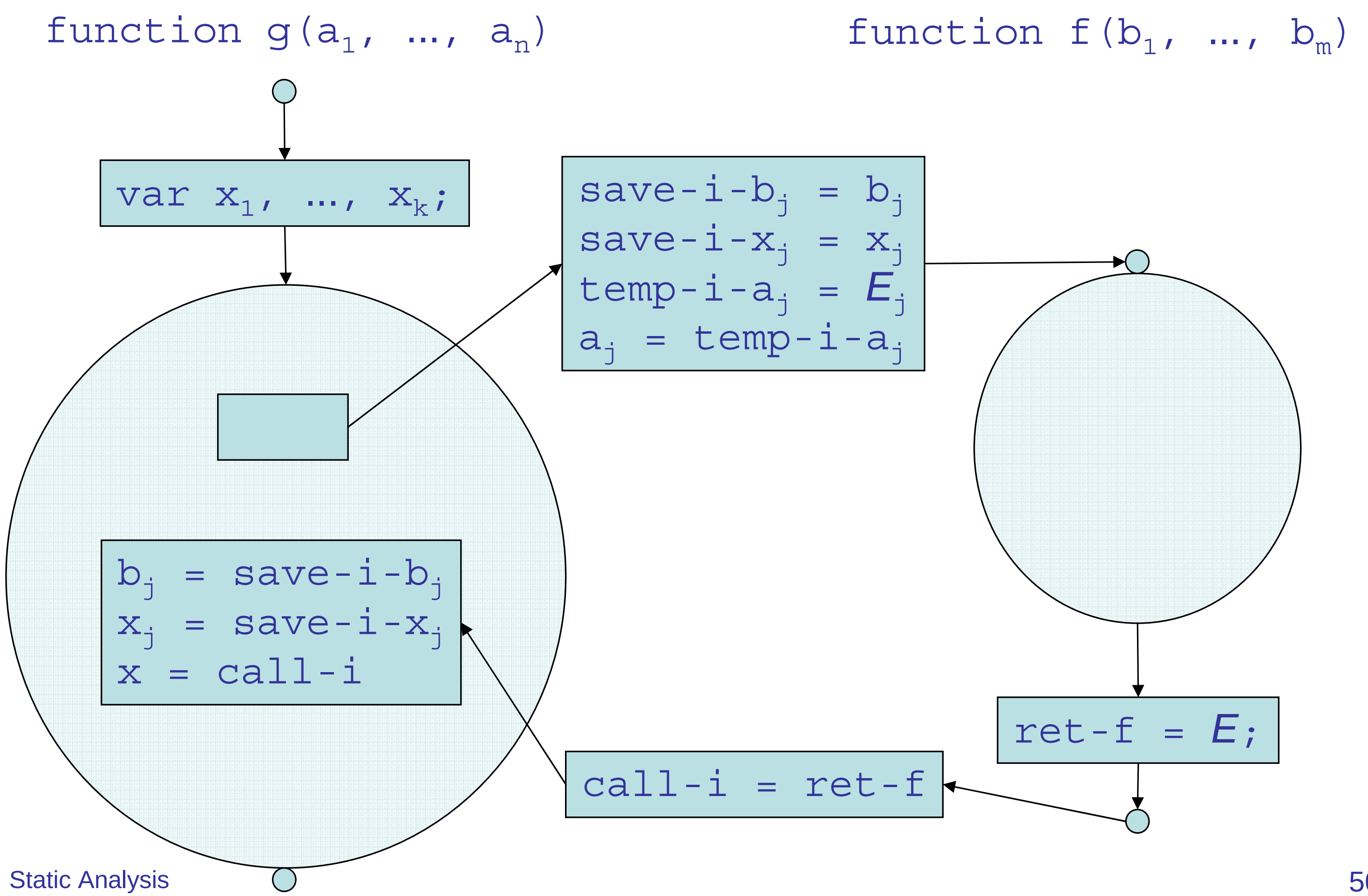
Shadow Variables

- Introduce some extra variables in the program
- For every function `f` the variable `ret-f` denoting its return value
- For every call site with index `i` a variable `call-i` denoting the computed value
- For every local or formal `x` and call site with index `i` a register `save-i-x`
- For every formal `x` and every call site with index `i` a temporary variable `temp-i-x`

Calling and Called Function



Glued Together

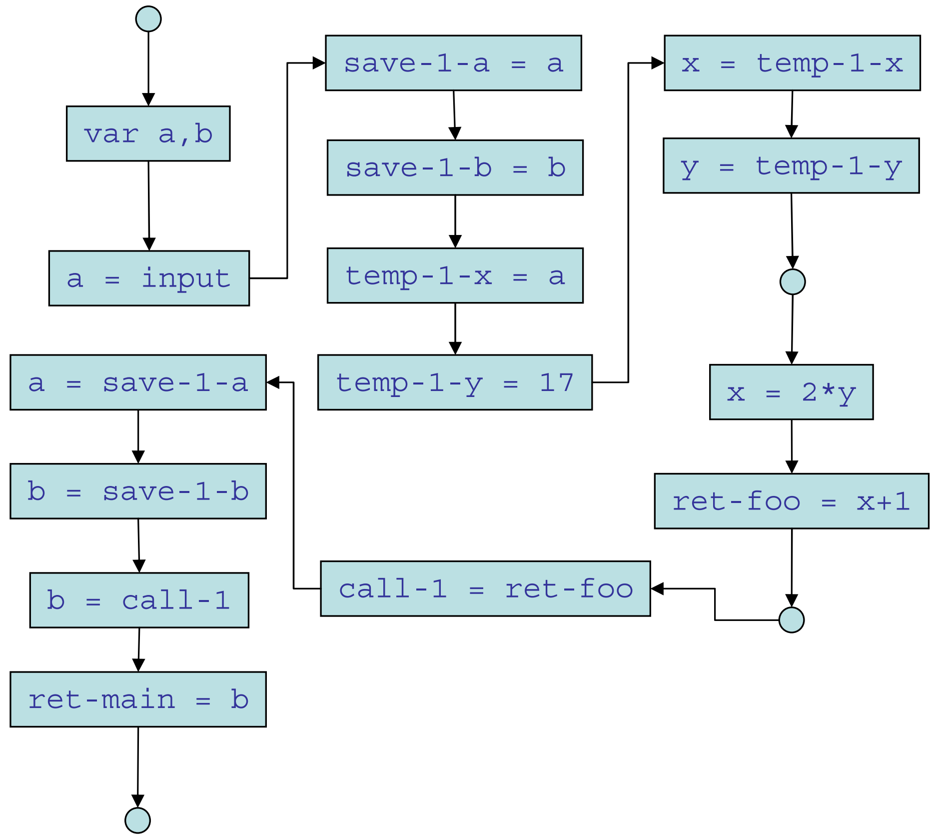


Example Program

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

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

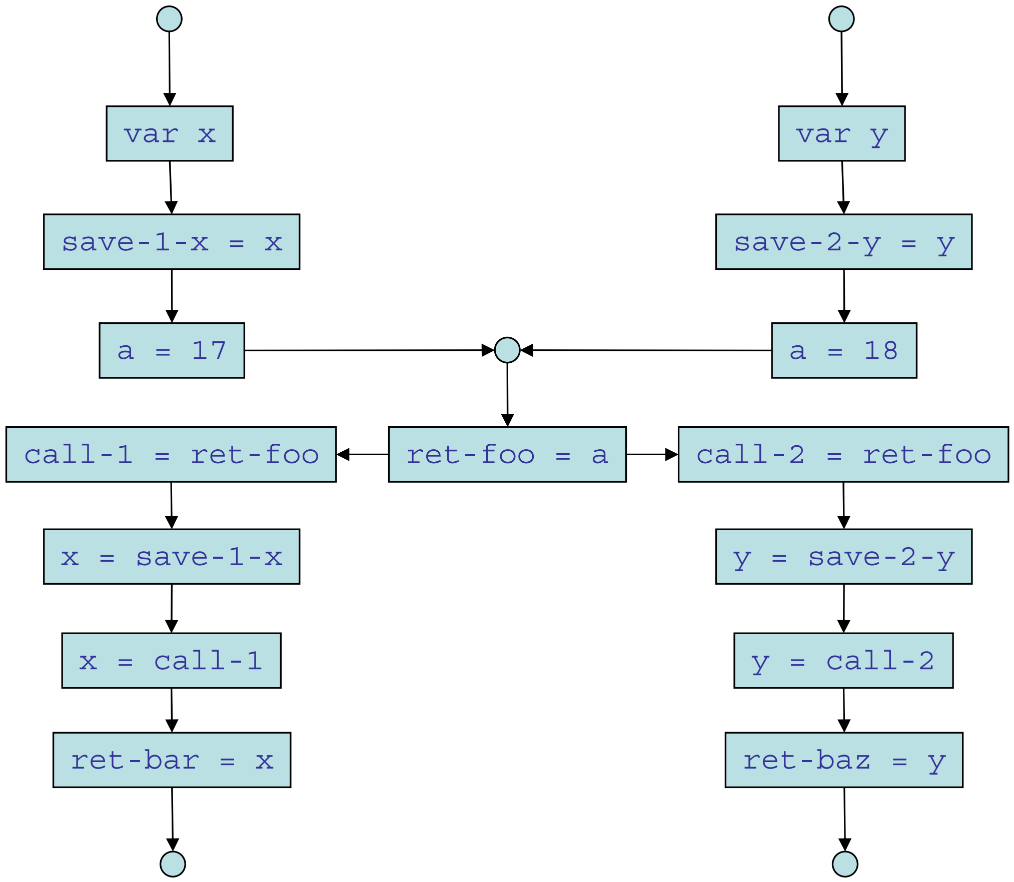
Resulting CFG

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

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



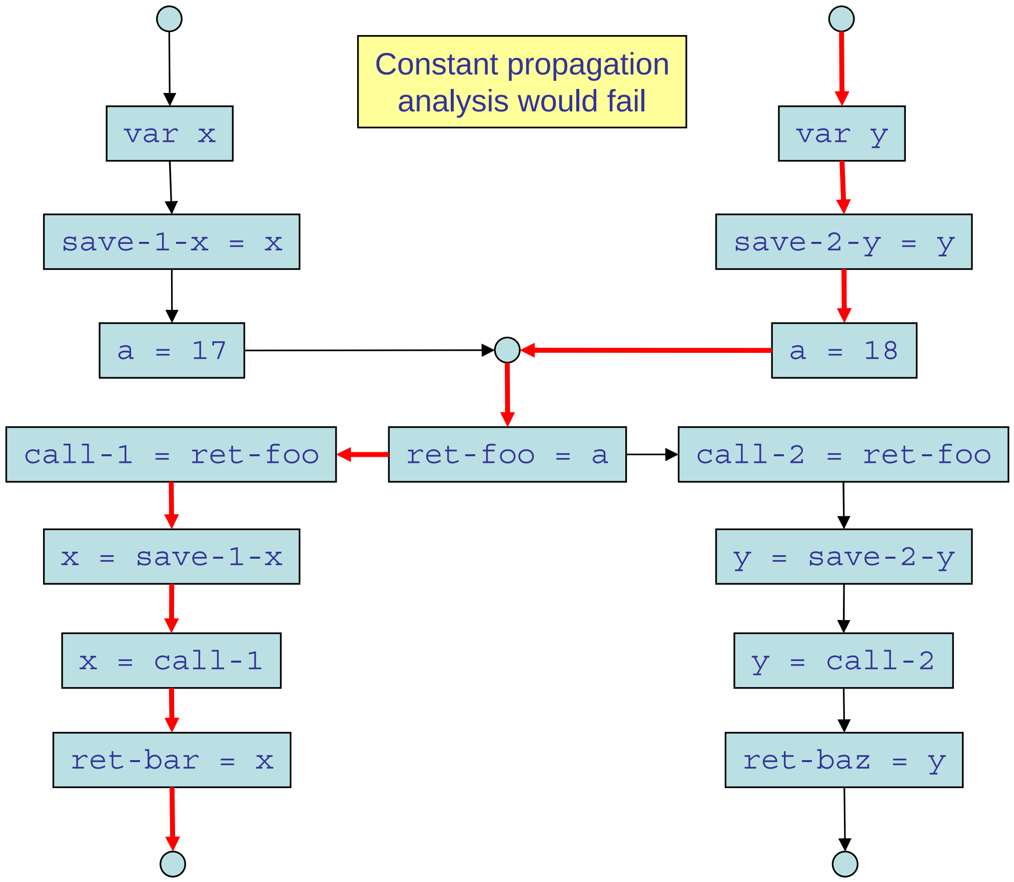
False Control Flow

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

bar() {
  var x;
  x = foo(17);
  return x;
}

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



False Control Flow

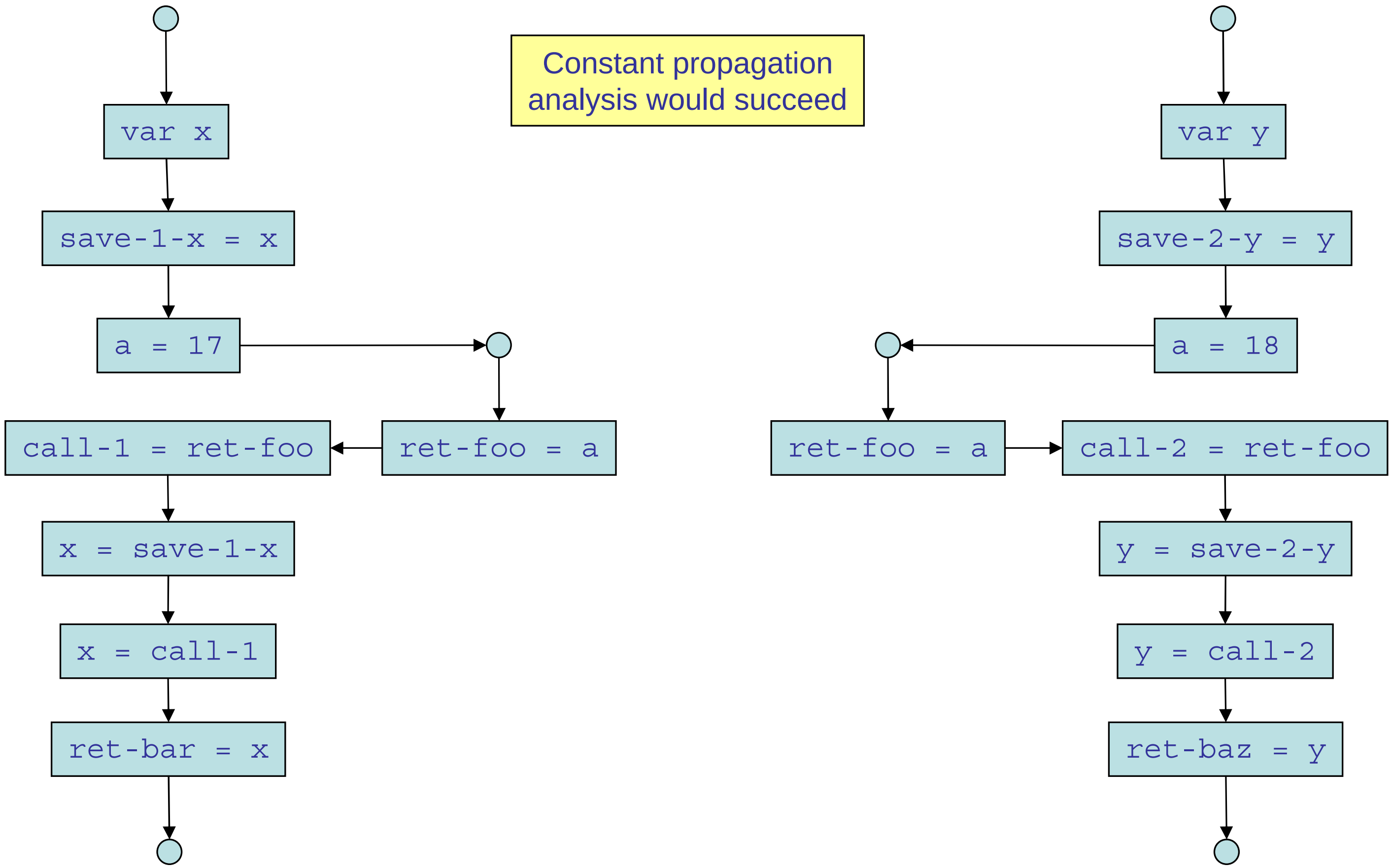
```
foo(a) {
  return a;
}

bar() {
  var x;
  x = foo(17);
  return x;
}

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



Polyvariance vs. Monovariance

- A *polyvariant* analysis creates *multiple copies* of the CFG for the body of a called function
- A *monovariant* analysis uses only one copy
- Strategies determine the number of copies:
  - the simplest is one copy for each call site
  - dynamic heuristics are also possible
  - important that only finitely many copies are created

Polyvariant CFG



Tree Shaking

- Identify those functions that are never called
  - safely remove them from the program
  - reduces size of the compiled executable
  - reduces size of CFG for subsequent analyses
- Uses monovariant interprocedural CFG
- Essentially a transitive closure computation

Setting Up

- The lattice is the powerset of all function names
- For every CFG node v we introduce a constraint variable [[v]] denoting the set of function that could *possibly* be called in the *future*
- We let *entry*(*id*) denote the entry node in the CFG for the function named *id*

Tree Shaking Constraints

- For assignments, conditions and `output`:

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

- For all other nodes:

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

- Here *funcs* is defined as:
  - $funcs(id) = funcs(intconst) = funcs(\texttt{input}) = \varnothing$
  - $funcs(E_1 \texttt{ op } E_2) = funcs(E_1) \cup funcs(E_2)$
  - $funcs(id\,(E_1, \ldots, E_n)) = \{id\} \cup funcs(E_i)$