Principles and applications of abstract-interpretation-based static analysis

David Schmidt

Kansas State University

www.cis.ksu.edu/~schmidt

Outline

Static analysis is property extraction from formal systems.

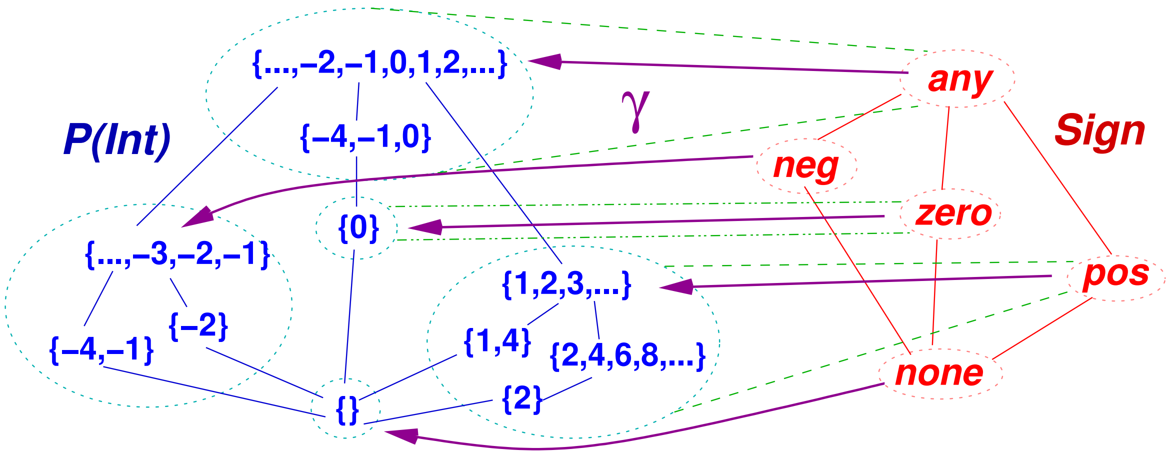
Abstract interpretation is a foundation for static analysis based on Galois connections, semi-homomorphisms, and fixed-point calculation. In this talk, we

- introduce abstract interpretation
- apply it to static analyses of program semantics (state-transition systems, equationally specified definitions, rule-based relational definitions)
- survey applications of static analysis
- develop the correspondence of properties to propositions
- consider approaches to modular, "scalable" analyses

Background: abstract interpretation

An abstract domain defines properties

A formal system uses values from set C, and we wish to determine properties of the C-values that might arise during computation.



Define an *abstract domain*, A: a partially ordered set of properties, closed under meet ($\sqcap$). See example, $Sign$, above.

Define a monotone *concretization map*, $\gamma : A \rightarrow PC$, where PC is the powerset of C, ordered by $\subseteq$, so that $\gamma(a)$ defines those elements that "have property a."

γ must *preserve meets* – for $T \subseteq A$, $\gamma(\sqcap T) = \bigcap_{a \in T} \gamma(a)$ – so that an inverse function, $\alpha : PC \rightarrow A$, can be defined.

Operations f *are abstracted to* $\mathsf{f}^\sharp$ *to compute on* A

```
readInt(x)
x = succ(x)
if x < 0 :
  x = negate(x)
else:
  x = succ(x)
writeInt(x)
```

Q:*is the output* pos?

A: abstractly interpret input domain Int by $Sign = \{neg, zero, pos, any, none\}$:

$\mathsf{readSign}(\mathtt{x})$
$\mathtt{x} = \mathsf{succ}^\sharp(\mathtt{x})$
$\mathtt{if}\ (\mathsf{filterNeg}(\mathtt{x})\!:$
$\quad \mathtt{x} = \mathsf{negate}^\sharp(\mathtt{x}))$
$\quad (\mathsf{filterNonNeg}(\mathtt{x})\!:$
$\quad \mathtt{x} = \mathsf{succ}^\sharp(\mathtt{x}))\ \mathtt{fi}$
$\mathsf{writeSign}(\mathtt{x})$

where

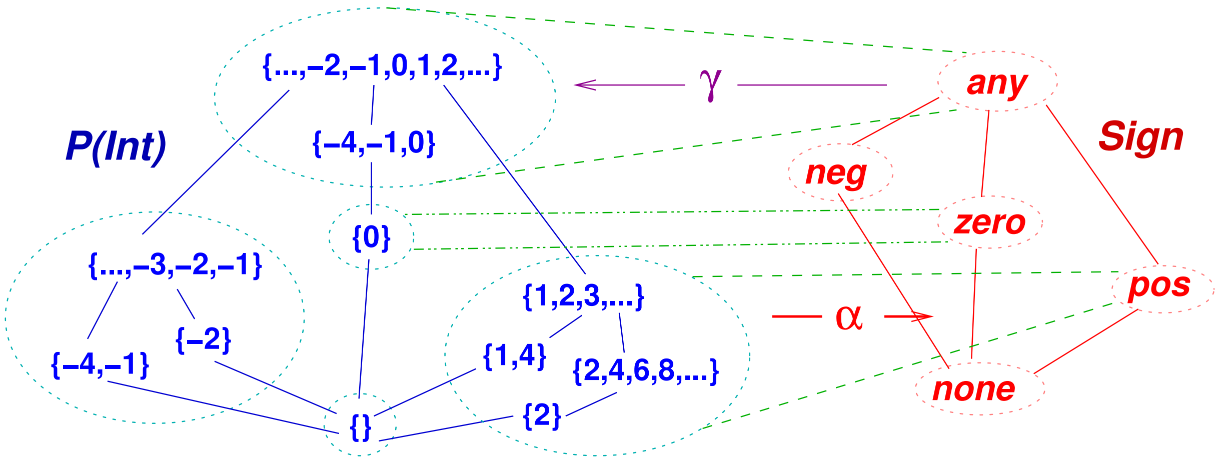
$$\begin{aligned} \mathsf{succ}^\sharp(pos) &= pos \\ \mathsf{succ}^\sharp(zero) &= pos \\ \mathsf{succ}^\sharp(neg) &= any \text{ (!)} \\ \mathsf{succ}^\sharp(any) &= any \end{aligned}$$

and

$$\begin{aligned} \mathsf{negate}^\sharp(neg) &= pos \\ \mathsf{negate}^\sharp(zero) &= zero \\ \mathsf{negate}^\sharp(pos) &= neg \\ \mathsf{negate}^\sharp(any) &= any \end{aligned}$$

For the abstract data-test sets, $zero, neg, pos$***, we calculate:***
$\{zero \mapsto pos,\ pos \mapsto pos,\ neg \mapsto any\}$. The last result arises because $\mathsf{succ}^\sharp(neg) = any$ and $\mathsf{filterNeg}(any) = neg$ (good!) but $\mathsf{filterNonNeg}(any) = any$ (bad — we need $zero \vee pos$!), so we cannot ensure the success of the else-arm.

A Galois connection *formalizes the abstraction*



$\gamma : Sign \rightarrow \mathcal{P}(Int)$

$\gamma(none) = \{\}, \quad \gamma(any) = Int$

$\gamma(neg) = \{\cdots, -3, -2, -1\}$

$\gamma(zero) = \{0\}, \quad \gamma(pos) = \{1, 2, 3, \cdots\}$

$\alpha : \mathcal{P}(Int) \rightarrow Sign$

$\alpha(S) = \sqcap\{a \mid \gamma(a) \subseteq S\}$

e.g., $\alpha\{2, 4, 6, 8, ...\} = pos$,

$\alpha\{-1, 0\} = any, \ \ \alpha\{0\} = zero$

$(\mathcal{P}(\mathrm{Int}), \subseteq)\langle\alpha, \gamma\rangle(Sign, \sqsubseteq)$ is a *Galois connection*:

$$\alpha(S) \sqsubseteq a \text{ iff } S \subseteq \gamma(a).$$

γ interprets the elements in $Sign$, and α maps each data-test set in $\mathcal{P}(Int)$ to the property that best describes the set [CousotCousot77] .

An abstract operation is monotone and sound

$f^\sharp : A \rightarrow A$ is *sound* for $f : C \rightarrow PC$ iff $\alpha \circ f^* \sqsubseteq f^\sharp \circ \alpha$
(iff $f^* \circ \gamma \sqsubseteq \gamma \circ f^\sharp$):

$$\begin{tikzcd} S \arrow[r, "f^*"] \arrow[d, "\alpha"'] & f^*[S] \arrow[d, "\alpha"] \\ \alpha(S) \arrow[r, "f^\#"] & \sqcap\!| \\ & \bullet \end{tikzcd} \qquad \begin{tikzcd} & \bullet \\ \gamma(a) \arrow[ru, "f^*"] & \sqcup\!| \\ a \arrow[u, "\gamma"] \arrow[r, "f^\#"] & f^\#(a) \arrow[u, "\gamma"'] \end{tikzcd}$$

α and γ act as *semi-homomorphisms*.

Example: The $succ^\sharp$ function seen earlier is sound for $succ$, e.g., for $succ : Int \rightarrow \mathcal{P}(Int)$, $succ^*(0) = \{1\}$, and $succ^\sharp(zero) = pos$.

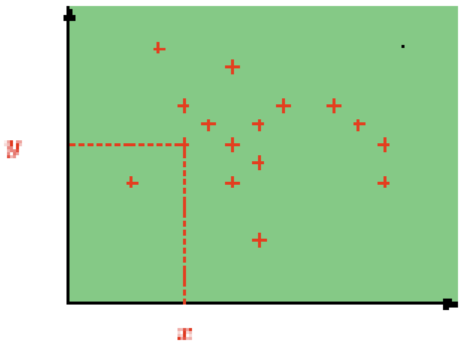
$f^\sharp$ is a *postcondition transformer*: $S \subseteq \gamma(a)$ implies $f^*(S) \subseteq \gamma(f^\sharp(a))$ where $f^*(S) = \cup_{c \in S} f(c)$.

$f^\sharp_{best} = \alpha \circ f^* \circ \gamma$ is the *strongest (liberal) postcondition* transformer.

Definition: $f^\sharp$ is γ*-complete ("forwards complete") for* f iff $f^* \circ \gamma = \gamma \circ f^\sharp$ [Giacobazzi01] . $f^\sharp$ is α*-complete ("backwards complete") for* f iff $\alpha \circ f^* = f^\sharp \circ \alpha$ [Cousots00] .

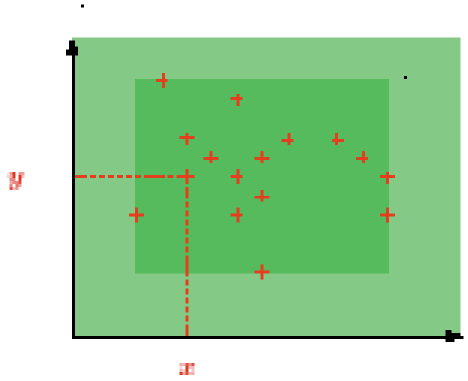
An aggregate, e.g., $Var \to \mathrm{C}$, can be abstracted pointwise or relationally

Sign: $[\mathtt{x} \mapsto \geq 0][\mathtt{y} \mapsto \geq 0]$



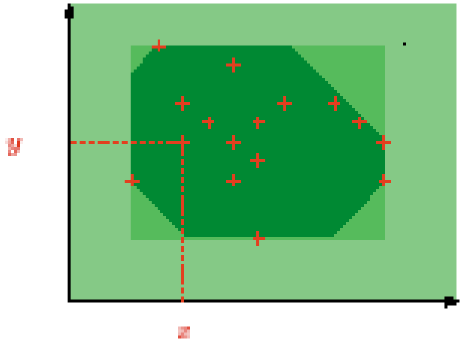
$$\begin{cases} x \geq 0 \\ y \geq 0 \end{cases}$$

Interval: $[\mathtt{x} \mapsto [3, 27]][[\mathtt{y} \mapsto [4, 32]]$



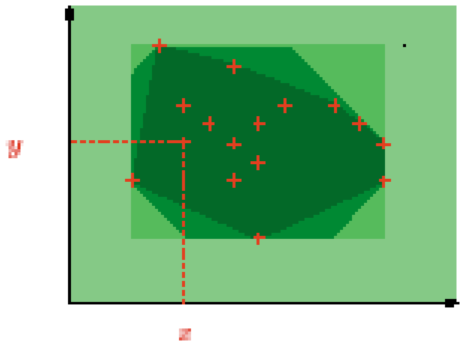
$$\begin{cases} x \in [3,\ 27] \\ y \in [4,\ 32] \end{cases}$$

Octagon: $\bigwedge_i (\pm \mathtt{x}_i \pm \mathtt{y}_i \leq c_i)$



$$\begin{cases} 3 \leq x \leq 27 \\ x + y \leq 88 \\ 4 \leq y \leq 32 \\ x - y \leq 61 \end{cases}$$

Polyhedron: $\bigwedge_i ((\sum_j a_{ij} \cdot \mathtt{x}_{ij}) \leq b_i)$



$$\begin{cases} 7x + 31y \leq 325 \\ 21x + 7y \geq 0 \end{cases}$$

diagrams from *Abstract Interpretation: Achievements and Perspectives* by Patrick Cousot, Proc. SSGRR 2000.

Three codings (a)-(c) of a relationally abstracted store based on the octagon abstract domain:

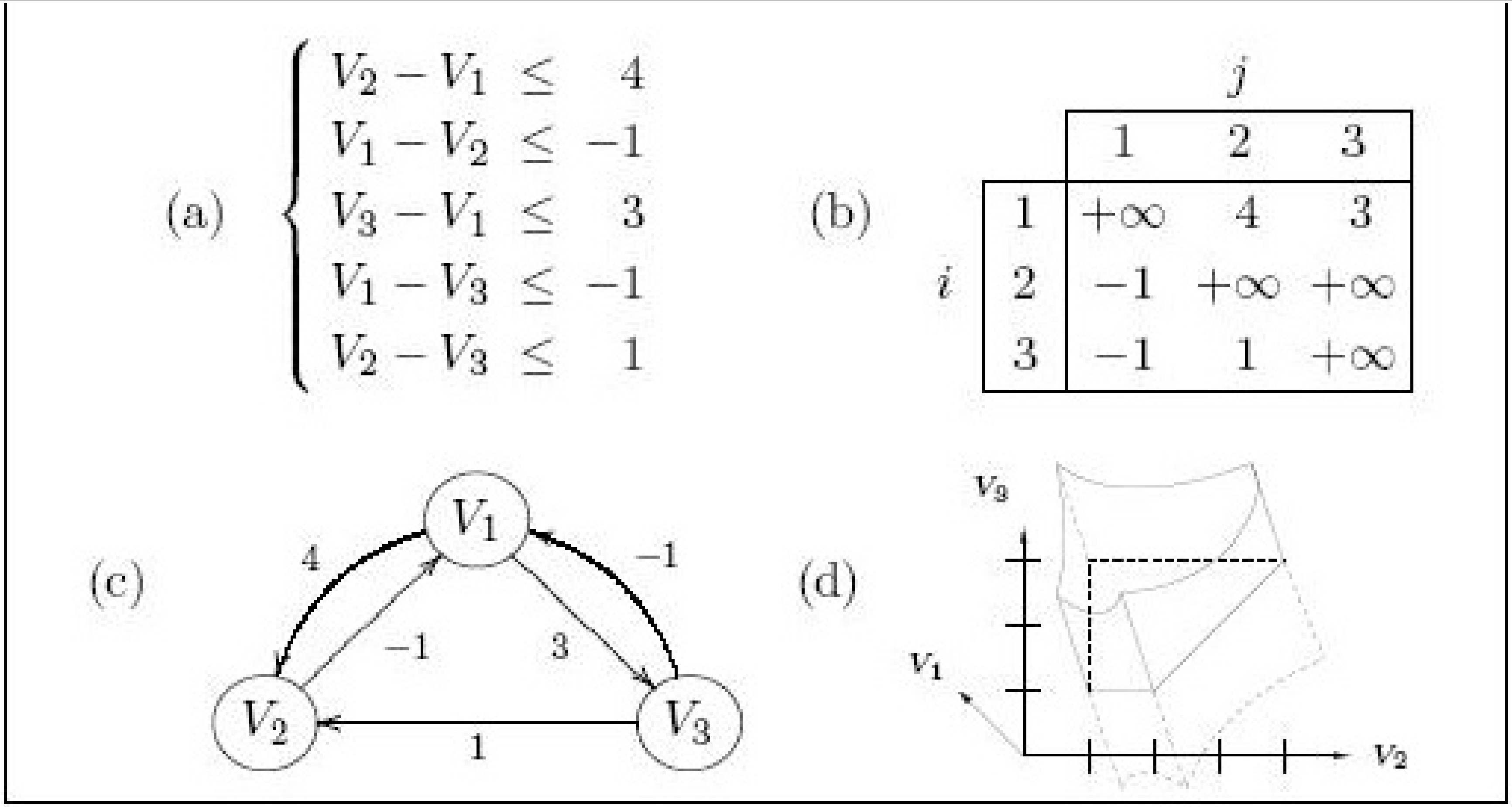


Figure 2. A potential constraint conjunction (a), its corresponding DBM $\mathbf{m}$ (b), potential graph $\mathcal{G}(\mathbf{m})$ (c), and potential set concretization $\gamma^{Pot}(\mathbf{m})$ (d).

diagram from *The octagon abstract domain*, by Antoine Miné, *J. Symbolic and Higher-Order Computation* 2006

Predicate abstraction *uses a relational domain based on the predicates in the goal and program*

Example: prove that $z \geq x \wedge z \geq y$ at p_3:

p_0 : `if x < y`
p_1 : `  then z = y`
p_2 : `  else z = x`
p_3 : ***exit***

$$\begin{tikzcd} & p_0, \langle ?, ?, ? \rangle \arrow[dl] \arrow[dr] & \\ p_1, \langle t, ?, ? \rangle \arrow[d] & & p_2, \langle f, ?, ? \rangle \arrow[d] \\ p_3, \langle t, t, t \rangle & & p_3, \langle f, t, t \rangle \end{tikzcd}$$

The store is abstracted to a relational domain that denotes the values of these predicates:

$$\phi_1 = x < y \qquad \phi_2 = z \geq x \qquad \phi_2 = z \geq y$$

The predicates are evaluated at the program's points as one of $\{t, f, ?\}$. (Read $?$ as $t \vee f$.)

At all occurrences of p_3 in the abstract trace, $\phi_2 \wedge \phi_3$ holds.

When a goal is undecided, **domain refinement** *becomes necessary*

Prove $\phi_0 \equiv \mathtt{x} \geq \mathtt{y}$ at p_4:

```
p0 :  if !(x >= y)
p1 :  then { i = x;
         p2 :  x = y;
         p3 :  y = i;
p4 :  }
```

$$
\begin{tikzcd}
 & p_0, \langle ? \rangle \arrow[dl] \arrow[dr] & \\
p_1, \langle f \rangle & & p_4, \langle t \rangle \\
p_2, \langle f \rangle & & \\
p_3, \langle t \rangle & & \\
p_4, \langle ? \rangle & &
\end{tikzcd}
$$

To decide the goal, we refine the abstract domain by adding a new predicate: $wp(\mathtt{y} = \mathtt{i}, \mathtt{x} \geq \mathtt{y}) = (\mathtt{x} \geq \mathtt{i}) \equiv \phi_1$. We add ϕ_1 and try again:

$$
\begin{tikzcd}
 & p_0, \langle ?, ? \rangle \arrow[dl] \arrow[dr] & \\
p_1, \langle f, ? \rangle & & p_4, \langle t \rangle \\
p_2, \langle f, t \rangle & & \\
p_3, \langle t, t \rangle & & \\
p_4, \langle t, t \rangle & &
\end{tikzcd}
$$

because $\mathtt{x} \not\geq \mathtt{y}$ and $\mathtt{x} \geq \mathtt{i}$ imply $\mathtt{y} > \mathtt{i}$ implies $\mathtt{x}_{new} \geq \mathtt{i}$

But incremental predicate refinement cannot synthesize many interesting loop invariants. For this example:

```
p0 : i = n; x = 0;
p1 : while  i != 0  {
  p2 : x = x + 1;  i = i - 1;
     }
p3 : goal: x = n
```

The initial predicate set, $P_0 \equiv \{i = 0, x = n\}$, does not validate the loop body.

The first refinement suggests we add $P_1 \equiv \{i = 1, x = n - 1\}$ to the program state, but this fails to validate a loop that iterates more than once.

Refinement stage j adds predicates $P_j \equiv \{i = j, x = n - j\}$; the refinement process continues forever!

The loop invariant is $x = n - i$:-)

Mechanics of static analysis: abstracting small-step and big-step semantics definitions

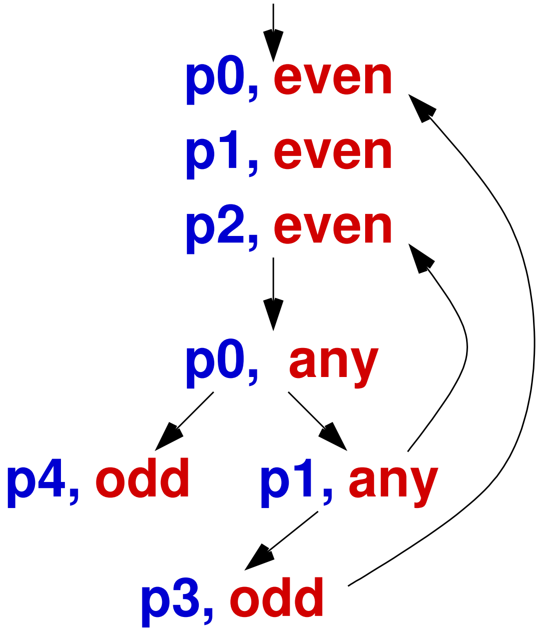
The most basic static analysis is **trace generation**

```
p0 :  while (x != 1) {
  p1 : if Even(x)
    p2 : then  x = x div2;
    p3 : else  x = 3*x + 1;
  }
p4 :  exit
```

Two concrete traces:

$\downarrow$	$\downarrow$
$p_0, 4$	$p_0, 6$
$p_1, 4$	$p_1, 6$
$p_2, 4$	$p_2, 6$
$\downarrow$	$\downarrow$
$p_0, 2$	$p_0, 3$
$p_1, 2$	$p_1, 3$
$p_2, 2$	$p_2, 3$
$\downarrow$	$\downarrow$
$p_0, 1$	$p_0, 10$
$p_4, 1$	$\cdots$
	$\downarrow$
	$p_4, 1$

Note: p_i, v abbreviates $p_i, \langle x : v \rangle$



For $Parity = \{none, even, odd, any\}$, the loop's operations, f, are abstracted to $f^\sharp$. The *abstract trace* is a static analysis of those concrete executions with an even-valued input. Traces are used in *model checking.*

Data-flow analysis *collects the abstract trace into a map,* $\mathrm{ProgramPoint} \rightarrow A$

The abstract value "attached" to program point p_i is defined by the first-order equational pattern,

$$p_i\mathrm{Store} = \bigsqcup_{p_j \in pred(p_i)} f_j^{\#}(p_j\mathrm{Store})$$

Flow equations for previous example:

$$\mathrm{init} = \langle x{:}\ \mathit{even} \rangle$$

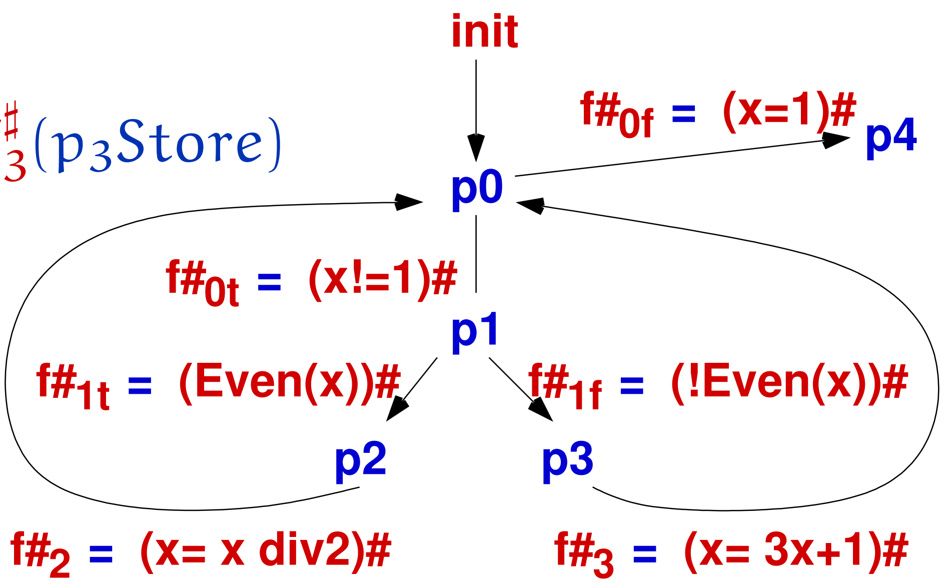
$$p_0\mathrm{Store} = \mathrm{init} \sqcup f_2^{\sharp}(p_2\mathrm{Store}) \sqcup f_3^{\sharp}(p_3\mathrm{Store})$$

$$p_1\mathrm{Store} = f_{0t}^{\sharp}(p_0\mathrm{Store})$$

$$p_2\mathrm{Store} = f_{1t}^{\sharp}(p_1\mathrm{Store})$$

$$p_3\mathrm{Store} = f_{1f}^{\sharp}(p_1\mathrm{Store})$$

$$p_4\mathrm{Store} = f_{0f}^{\sharp}(p_0\mathrm{Store})$$



We *solve* the flow equations by calculating approximate solutions in stages until *the least fixed point* is reached.

Note: $\perp$ is the same as $\langle \mathrm{x}{:}\perp \rangle$

stage	p_0Store	p_1Store	p_2Store	p_3Store	p_4Store
0	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
1	$\langle \mathrm{x}{:}\,even \rangle$	$\perp$	$\perp$	$\perp$	$\perp$
2	$\langle \mathrm{x}{:}\,even \rangle$	$\langle \mathrm{x}{:}\,even \rangle$	$\perp$	$\perp$	$\perp$
3	$\langle \mathrm{x}{:}\,even \rangle$	$\langle \mathrm{x}{:}\,even \rangle$	$\langle \mathrm{x}{:}\,even \rangle$	$\perp$	$\perp$
4	$\langle \mathrm{x}{:}\,any \rangle$	$\langle \mathrm{x}{:}\,even \rangle$	$\langle \mathrm{x}{:}\,even \rangle$	$\perp$	$\perp$
5	$\langle \mathrm{x}{:}\,any \rangle$	$\langle \mathrm{x}{:}\,any \rangle$	$\langle \mathrm{x}{:}\,even \rangle$	$\perp$	$\langle \mathrm{x}{:}\,odd \rangle$
6	$\langle \mathrm{x}{:}\,any \rangle$	$\langle \mathrm{x}{:}\,any \rangle$	$\langle \mathrm{x}{:}\,even \rangle$	$\langle \mathrm{x}{:}\,odd \rangle$	$\langle \mathrm{x}{:}\,odd \rangle$

A faster algorithm uses a *worklist* that remembers exactly which equations should be recalculated at each stage.

Termination: *Array-bounds checking reviewed*

Integer variables might receive values from the *interval domain*,

$$I = \{[i, j] \mid i, j \in \mathrm{Int} \cup \{-\infty, +\infty\}\}.$$

We define $[a, b] \sqcup [a', b'] = [\min(a, a'), \max(b, b')]$.

```
int a = new int[10];
i = 0;
while (i < 10)  {
      ... a[i] ...
      i = i + 1;
}
```

i = [0,0]

p_1
i = [0,0] $\sqcap$ *[− ∞ ,9] = [0,0]*
i = [0,0] $\sqcup$ *[1,1]* $\sqcap$ *[− ∞ ,9] = [0,1]*
...

p_2
i = [1,1]
i = [1,1] $\sqcup$ *[2,2] = [1,2]*
...

This example terminates: `i`’s ranges are

at p_1 : $[0..9]$

at p_2 : $[1..10]$

at loop exit : $[1..10] \sqcap [10, +\infty] = [10, 10]$

But others might not, because the domain is not finite height:

```
i = 0;
while true {
   ...
   i = i + 1;
}
```

i = [0,0]

i = [0,0] $\sqcup$ [1,1] $\sqcup$ [2, 2] ...

infinite limit is [0, + ∞]

i = [ ]   (dead code)

The analysis generates the infinite sequence of stages, $[0, 0], [0, 1], ..., [0, i], ...$ as `i`'s value in the loop's body.

The domain of intervals, where $[i, j] \sqsubseteq [i', j']$ *iff* $i \leq j$ *and* $j \leq j'$*, has infinitely ascending chains.*

To forcefully terminate the analysis, we can replace the $\sqcup$ operation by ∇, called a *widening operator*:

$$[\,]\nabla[i, j] = [i, j] \qquad [i, j]\nabla[i', j'] = \begin{array}{l}[\text{if } i' < i \text{ then } -\infty \text{ else } i, \\ \text{if } j' > j \text{ then } +\infty \text{ else } j]\end{array}$$

The widening operator, which guarantees finite convergence for all increasing sequences on the interval domain, quickly terminates the example:

```
i = 0;                      i = [0,0]
while true {
     ...                    i = [0,0] ∇ [1,1] = [ 0, + ∞ ]
   i = i + 1;
}
                            i = [ ]   (dead code)
```

but in general, it can lose much precision:

```
int a = new int[10];
i = 0;                      i = [0,0]
while (i < 10)  {
     ... a[i]               i = [0,0] ∇ [1,1] = [ 0, + ∞ ]
     i = i + 1;
}
                            i = [ 10, + ∞ ]
```

For this reason, a complementary operation, $\triangle$, called a *narrowing operation*, can be used after ∇ gives convergence to recover some precision and retain a fixed-point solution.

We will not develop $\triangle$ here, but for the interval domain, a suitable $\triangle$ tries to reduce $-\infty$ and $+\infty$ to finite values. For the last example, the convergent value, $[0, +\infty]$, in the loop body would be narrowed to $[0, 10]$, making `i`'s value on loop exit $[10, 10]$.

Another approach is to use multiple "thresholds" for widening, e.g. $-\infty$, $(2^{-31} - 1)$, 0, etc. for lower limits, and $(2^{31} - 1)$ and $+\infty$ for upper limits.

Structured static analysis on syntax trees

Given a block of statements, B, we might wish to calculate the values that "enter" and "exit" from B. If B is coded in a structured language, the static analysis can compute a "structured transfer function" for B:

$$\mathrm{C} \ ::= \ \mathrm{p} : \mathtt{x} \ = \mathrm{E} \mid \mathrm{C} \mid \mathtt{if}\ \mathrm{E}\ \mathrm{C}_1\ \mathrm{C}_2 \mid \mathtt{while}\ \mathrm{E}\ \mathrm{C}$$

A sample structured analysis that ignores tests: $[\![\mathrm{C}]\!] : A_{\mathrm{in}} \to A_{\mathrm{out}}$

$[\![\mathrm{p} : \mathtt{x} \ = \mathrm{E}]\!]\mathrm{in} = f^{\sharp}_{\mathrm{p}}(\mathrm{in})$ (the transfer function for p)

$[\![\mathrm{C}]\!]\mathrm{in} = [\![\mathrm{C}_2]\!]([\![\mathrm{C}_1]\!]\mathrm{in})$

$[\![\mathtt{if}\ \mathrm{E}\ \mathrm{C}_1\ \mathrm{C}_2]\!]\mathrm{in} = [\![\mathrm{C}_1]\!]\mathrm{in} \sqcup [\![\mathrm{C}_2]\!]\mathrm{in}$

$[\![\mathtt{while}\ \mathrm{E}\ \mathrm{C}]\!]\mathrm{in} = \mathrm{in} \sqcup \mathrm{out}_{\mathrm{C}}$,

where $\mathrm{out}_{\mathrm{C}} = \bigsqcup_{i \geq 0} \mathrm{out}_i$,

and $\mathrm{out}_0 = \perp_A$ and $\mathrm{out}_{i+1} = [\![\mathrm{C}]\!](\mathrm{in} \sqcup \mathrm{out}_i)$

We annotate a syntax tree with the in-and out-data — here is a reaching definitions data-flow analysis, which computes sets of assignments that might reach future program points:

$$[\![p : \mathtt{x} \;= E]\!]\mathit{in} = \mathit{in} - \mathit{kill}_{\mathtt{x}} \cup \{p\}$$

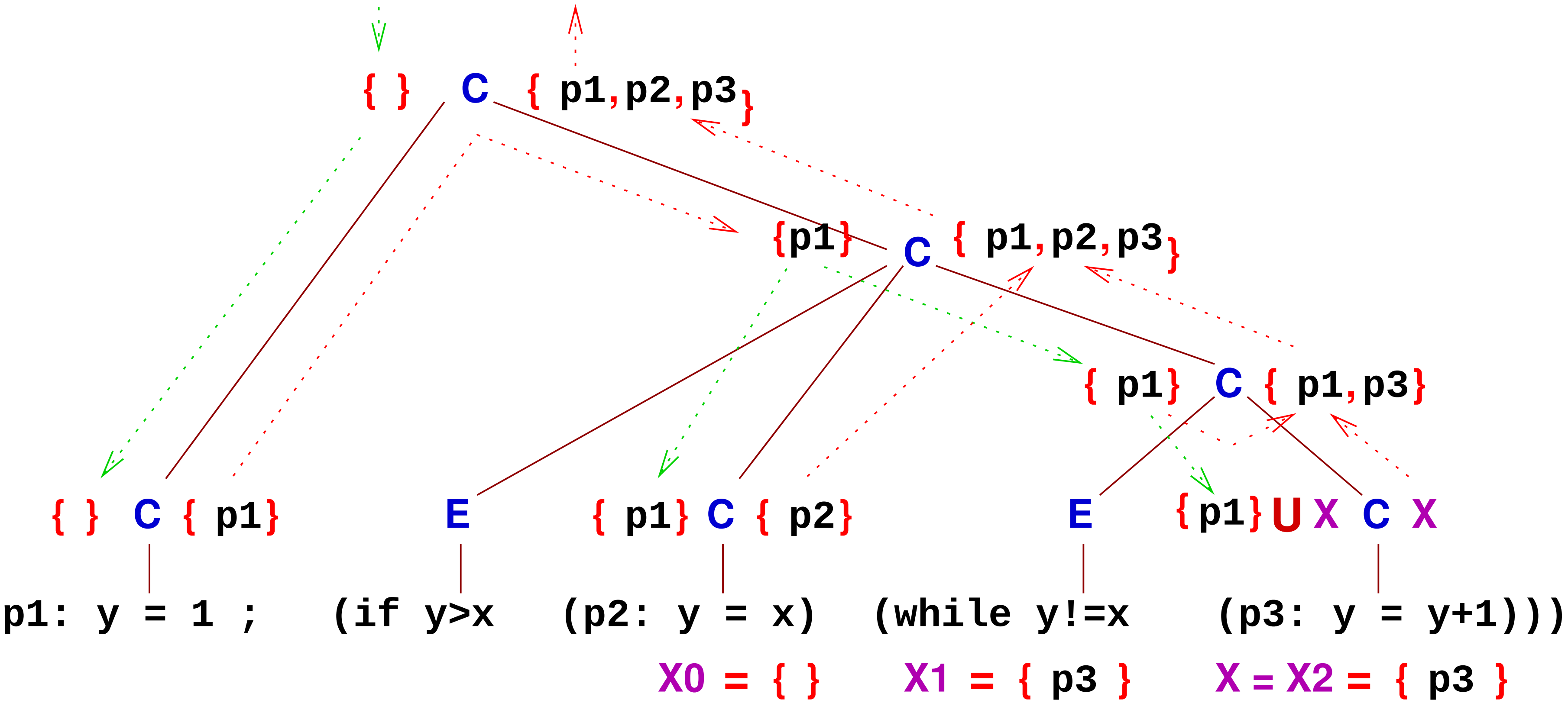
$$[\![C]\!]\mathit{in} = [\![C_2]\!]([\![C_1]\!]\mathit{in})$$

$$[\![\mathtt{if}\ E\ C_1\ \ C_2]\!]\mathit{in} = [\![C_1]\!]\mathit{in}\ \cup\ [\![C_2]\!]\mathit{in}$$

$$[\![\mathtt{while}\ E\ C]\!]\mathit{in} = \mathit{in} \cup \bigcup_{i\geq 0} \mathit{out}_i,$$

where $\mathit{out}_0 = \{\}$

and $\mathit{out}_{i+1} = [\![C]\!](\mathit{in} \cup \mathit{out}_i)$



Big-step relational semantics: derivation trees

$$\sigma \vdash p : \mathtt{x} \;= E \Downarrow f_p(\sigma)$$

$$\frac{\sigma \vdash C_1 \Downarrow \sigma_1 \quad \sigma_1 \vdash C_2 \Downarrow \sigma_2}{\sigma \vdash C_1; C_2 \Downarrow \sigma_2} \qquad \frac{f_{Et}(\sigma) \vdash C_1 \Downarrow \sigma_1 \quad f_{Ef}(\sigma) \vdash C_2 \Downarrow \sigma_2}{\sigma \vdash \mathtt{if}\ E\ C_1\ C_2 \Downarrow \sigma_1 \sqcup \sigma_2}$$

$$\frac{f_{Et}(\sigma) \vdash C \Downarrow \sigma' \quad \sigma' \vdash \mathtt{while}\ E\ C \Downarrow \sigma''}{\sigma \vdash \mathtt{while}\ E\ C \Downarrow f_{Ef}(\sigma) \sqcup \sigma''} \qquad \bot \vdash C \Downarrow \bot$$

Recall that f_p is a transfer function and that f_{Et} and f_{Ef} ”filter” the store, e.g.,

$f_{\mathtt{x}>2t}\langle x:4, y:3\rangle = \langle x:4, y:3\rangle$, whereas $f_{\mathtt{x}>2t}\langle x:0, y:3\rangle = \bot$.

An example: `if Even(x) (x=0) (while x≠3 (x = x+1))`

$$\langle x{:}1\rangle \vdash \mathtt{if\ Even(x)\ (x = 0)\ (while\ x \neq 3\ (x = x + 1))} \Downarrow \bot \sqcup \langle x{:}3\rangle = \langle x{:}3\rangle$$

$$\bot \vdash \mathtt{x} = 0 \Downarrow \bot \qquad \langle x{:}1\rangle \vdash \mathtt{while\ x} \neq 3 \ldots \Downarrow \bot \sqcup \langle x{:}3\rangle$$

$$\langle x{:}1\rangle \vdash \mathtt{x} = \mathtt{x} + 1 \Downarrow \langle x{:}2\rangle \qquad \langle x{:}2\rangle \vdash \mathtt{while\ x} \neq 3 \ldots \Downarrow \bot \sqcup \langle x{:}3\rangle$$

$$\langle x{:}2\rangle \vdash \mathtt{x} = \mathtt{x} + 1 \Downarrow \langle x{:}3\rangle \qquad \langle x{:}3\rangle \vdash \mathtt{while\ x} \neq 3 \ldots \Downarrow \langle x{:}3\rangle \sqcup \bot = \langle x{:}3\rangle$$

$$\bot \vdash \mathtt{x} = \mathtt{x} + 1 \Downarrow \bot \qquad \bot \vdash \mathtt{while\ x} \neq 3 \ldots \Downarrow \bot$$

An abstract big-step derivation tree

Using the same inference rules but with abstract transfer functions for $\mathrm{Parity} = \{\bot, \mathit{even}, \mathrm{odd}, \top\}$, we generate an abstract tree that is *infinite* but *regular*:

$$\langle \mathrm{x{:}odd} \rangle \vdash \texttt{if Even(x) (x = 0) (while x} \neq \texttt{3 (x = x + 1))} \Downarrow \bot \sqcup \mathsf{X}$$

$$\bot \vdash \texttt{x} = \texttt{0} \Downarrow \bot \qquad \langle \mathrm{x{:}odd} \rangle \vdash \texttt{while x} \neq \texttt{3...} \Downarrow \langle \mathrm{x{:}odd} \rangle \sqcup \mathsf{X} = \mathsf{X}$$

$$\langle \mathrm{x{:}odd} \rangle \vdash \texttt{x} = \texttt{x} + \texttt{1} \Downarrow \langle \mathrm{x{:}}\mathit{even} \rangle \qquad \langle \mathrm{x{:}}\mathit{even} \rangle \vdash \texttt{while x} \neq \texttt{3...} \Downarrow \bot \sqcup \mathsf{X}$$

$$\langle \mathrm{x{:}}\mathit{even} \rangle \vdash \texttt{x} = \texttt{x} + \texttt{1} \Downarrow \langle \mathrm{x{:}odd} \rangle \qquad \langle \mathrm{x{:}odd} \rangle \vdash \texttt{while x} \neq \texttt{3...} \Downarrow \mathsf{X}$$

$$\ldots$$

Variable X denotes the answer from the repeated loop subderivation:

$$\mathsf{X} = \langle \mathrm{x{:}odd} \rangle \sqcup \mathsf{X}$$

The least solution sets $\mathsf{X} = \langle \mathrm{x{:}odd} \rangle$.

Interprocedural analysis

$$\frac{\texttt{func f(x) local y; } C. \qquad [x \mapsto [\![E]\!]\sigma][y \mapsto \bot] \vdash C \Downarrow \sigma'}{\sigma \vdash \texttt{z} = \texttt{f}(E) \Downarrow \sigma[\texttt{z} \mapsto \sigma'(\texttt{y})]}$$

where $[\![E]\!]\sigma$ denotes E's value with σ, and $x \mapsto v$ assigns v to x.

Example:

```
func g(x) local z; z = x+1.
a = g(2); b = g(a); a = a*b
```

$$\langle a{:}\bot, b{:}\bot\rangle \vdash a = g(2);\ b = g(a);\ a = a * b \Downarrow \langle a{:}even, b{:}even\rangle$$

$$\langle a{:}\bot, b{:}\bot\rangle \vdash a = g(2) \Downarrow \langle a{:}odd, b{:}\bot\rangle \qquad \langle a{:}odd, b{:}\bot\rangle \vdash b = g(a);\ a = a * b \Downarrow \langle a{:}even, b{:}even\rangle$$

$$\langle x{:}even, z{:}\bot\rangle \vdash z = x + 1 \Downarrow \langle x{:}even, z{:}odd\rangle \qquad \langle a{:}odd, b{:}even\rangle \vdash a = a * b \Downarrow \langle a{:}even, b{:}even\rangle$$

$$\langle a{:}odd, b{:}\bot\rangle \vdash b = g(a); \Downarrow \langle a{:}odd, b{:}even\rangle$$

$$\langle x{:}odd, z{:}\bot\rangle \vdash z = x + 1 \Downarrow \langle x{:}odd, z{:}even\rangle$$

The derivation tree naturally separates the calling contexts.

"Too many" calling contexts (*) force widening (!):

$\texttt{func fac(a) local b; if a} = 0\ (\texttt{b} = 1)\ (\texttt{b} = \texttt{fac(a} - 1\texttt{); b} = \texttt{a} * \texttt{b}).$
$\texttt{c} = \texttt{fac(3)}$

$$\langle c:\bot\rangle \vdash \texttt{c} = \texttt{fac(3)} \Downarrow \langle c:\top\rangle$$

$$* \quad \langle 3,\bot\rangle \vdash \texttt{if a} = 0\ (\texttt{b} = 1)(\texttt{b} = \texttt{fac(a} - 1\texttt{); b} = \texttt{a} * \texttt{b}) \Downarrow \bot \sqcup \langle \top,\top\rangle = \langle \top,\top\rangle$$

$$\bot \vdash \texttt{b} = 1 \Downarrow \bot \qquad \langle 3,\bot\rangle \vdash \texttt{b} = \texttt{fac(a} - 1\texttt{); b} = \texttt{a} * \texttt{b} \Downarrow \langle \top,\top\rangle$$

$$\langle 3,\bot\rangle \vdash \texttt{b} = \texttt{fac(a} - 1) \Downarrow \langle 3,\top\rangle \qquad 3,\top \vdash \texttt{b} = \texttt{a} * \texttt{b} \Downarrow \top,\top$$

$$* \quad \langle 3,\bot\rangle \sqcup \langle 2,\bot\rangle = \langle \top,\bot\rangle \vdash \texttt{if a} = 0 \ldots \Downarrow \langle 0,1\rangle \sqcup \langle \top,\top * X.b\rangle = X = \langle \top,\top\rangle$$

$$\langle 0,\bot\rangle \vdash \texttt{b} = 1 \Downarrow \langle 0,1\rangle \qquad \langle \top,\bot\rangle \vdash \texttt{b} = \texttt{fac(a} - 1\texttt{); b} = \texttt{a} * \texttt{b} \Downarrow \langle \top,\top * X.b\rangle$$

$$\langle \top,\bot\rangle \vdash \texttt{b} = \texttt{fac(a} - 1) \Downarrow \langle \top, X.b\rangle \qquad \langle \top, X.b\rangle \vdash \texttt{b} = \texttt{a} * \texttt{b} \Downarrow \langle \top,\top * X.b\rangle$$

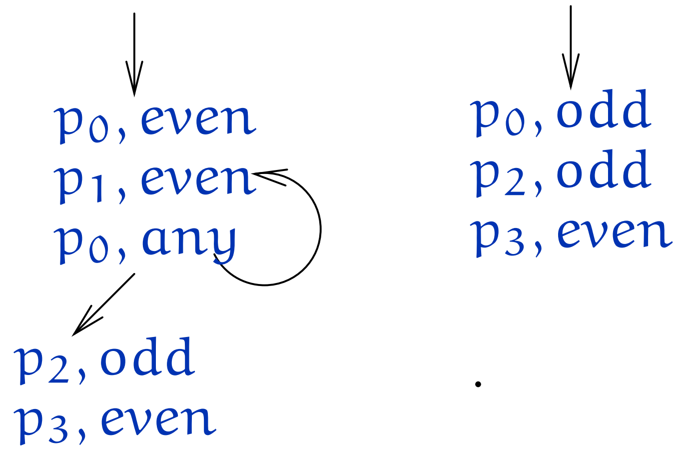
$$*! \quad \langle \top,\bot\rangle \vdash \texttt{if a} = 0 \ldots \Downarrow X$$

$X = \langle 0,1\rangle \sqcup \langle \top,\top * X.b\rangle$. **The least solution sets** $X = \langle \top,\top\rangle$.

Standard applications of static analysis

Abstract testing and model generation

```
p0 :  while isEven(x) {
        p1 :  x = x div 2;
      }
p2 :  x = 4 * x;
p3 :  exit
```



Each trace tree denotes an abstract "test" that covers a set of concrete test cases, e.g., $\gamma(even) = \{..., -2, 0, 2, ...\}$.

Forms of abstract testing:

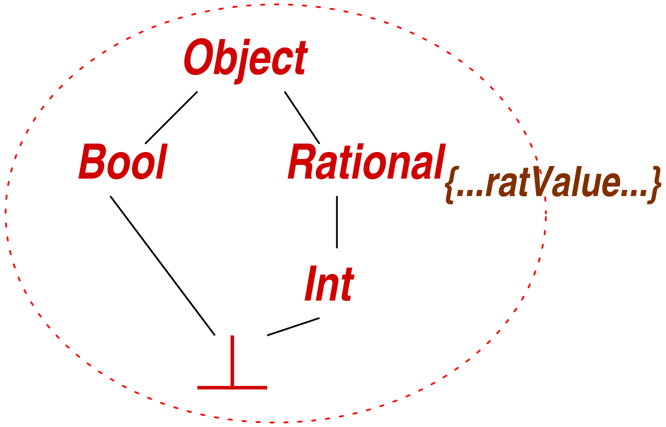
- *Black box*: For each test set, $S \subseteq C$, we abstractly interpret with $\alpha(S) \in A$. *(Best precision: ensure that $S = \gamma(\alpha(S))$.)*
- *White box:* for each conditional, B_i, in the program, ensure there is some $a_i \in A$ such that $\gamma(a_i) = \{s \mid B_i$ holds for $s\}$

Once we generate an abstract model, we can analyze it further — ask questions of its paths and nodes — via *model checking*.

Low-level safety checking

One example is *type casting*:

p_i : ... ((Rational) x).ratValue()...



A static analysis calculates the abstract store arriving at the cast at p_i, a *checkpoint*:

- $p_i, \langle ...x : \mathrm{Int}... \rangle$: no error possible — remove the run-time check (because $\mathrm{Int} \sqsubseteq \mathrm{Rational}$, hence $\gamma(\mathrm{Int}) \subseteq \gamma(\mathrm{Rational})$).
- $p_i, \langle ...x : \mathrm{Object}... \rangle$: possible error — retain run-time check (because $\mathrm{Object} \not\sqsubseteq \mathrm{Rational}$)
- $p_i, \langle ...x : \mathrm{Bool}... \rangle$: definite error, because $\mathrm{Bool} \sqcap \mathrm{Rational} = \bot$ (assuming $\gamma(\bot) = \{\}$).

Two more examples of low-level safety checking:

Array-bounds and arithmetic over- and under-flow checks

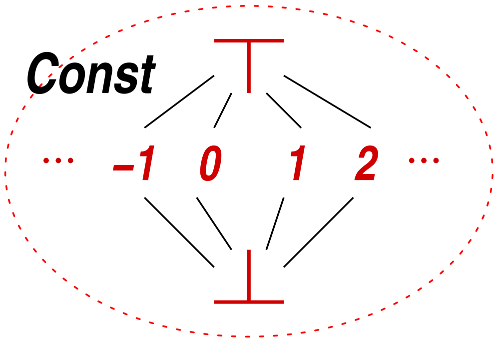
- *Analysis:* interval analysis, where values have form, $[i, j]$, $i \leq j$.
- *Checkpoints:* for `a[e]` — e has value in range, $[0, \text{a}.\mathrm{length}]$;
  for `int x = e` — e has value in range, $[-2^{31} - 1,\ +2^{31} - 1]$

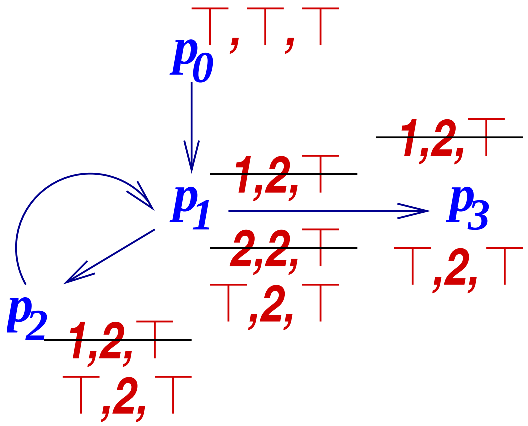
Uninitialized variables, dead-code, and erroneous-state checks

- *Analysis:* constant propagation, where values are $\{k\}$, $\perp$, or $\top$.
- *Checkpoints:*
  uninitialized variables: referenced variables have value $\neq \perp$;
  dead code: at program point p_i, arriving store has value $\neq \perp$;
  erroneous states: at program point $p_i : \mathrm{Error}$, arriving store has value $= \perp$. *(Note:* This can be combined with a *backwards* analysis, starting from each $p_i : \mathrm{Error}$ with store $\top$, working backwards to see if an initial state is reached.*)*

Program transformation: Constant folding

```
p0 :  x = 1; y = 2;
p1 :  while (x < y + z) {
        p2 : x = x + 1;
      }
p3 :  exit
```





The analysis tells us to replace $\mathtt{y}$ at p_1 by 2:

`x = 1; y = 2; while (x < 2 + z)  x = x + 1`

Basic principle of program transformation:

If $a_i \in A$ arrives at point $p_i : S$, where $f_i : C \to C$ is the concrete transfer function, and there are some S', f' such that $f_i(c) = f'(c)$ for all $c \sqsubseteq_C \gamma(a_i)$, then S can be replaced by S' at p_i.

For constant folding, the transformation criteria are the abstract integers $\ldots -1, 0, 1, \ldots$ (but not $\top$).

Precondition checking and assertion synthesis

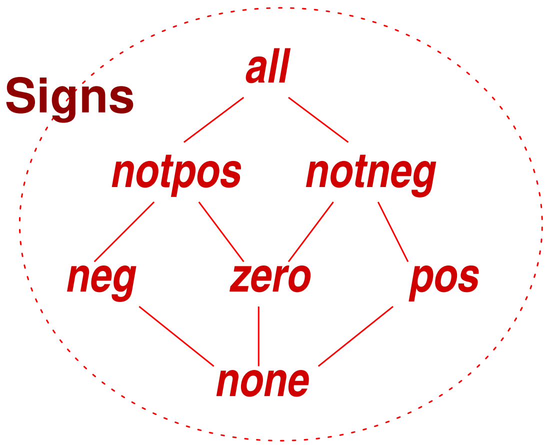
A backwards analysis synthesizes precondition assertions that ensure achievement of a postcondition:

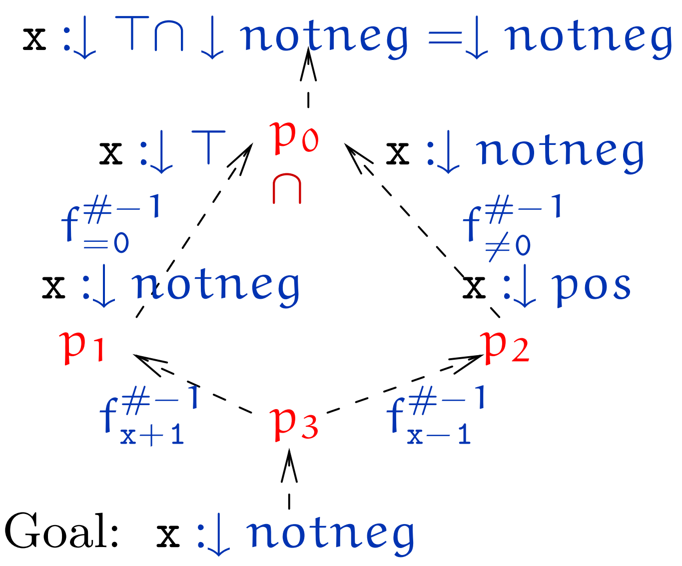
p_0: `if x=0`

$\quad p_1$: `then x = x+1`

$\quad p_2$: `else x = x-1`

p_3 : halt $\quad \langle \mathtt{x} :\downarrow notneg \rangle$





where

$f^\sharp_{=0}(a) = a \sqcap zero = \alpha \circ f_{=0} \circ \gamma$

$f^\sharp_{\neq 0} = \alpha \circ f_{\neq 0} \circ \gamma$, e.g., $f^\sharp_{\neq 0}(notneg) = pos$;

$\qquad f^\sharp_{\neq 0}(zero) = \bot$; $f^\sharp_{\neq 0}(\top) = \top$

$f^\sharp_{+1} = \alpha \circ f_{+1} \circ \gamma$, e.g., $f^\sharp_{+1}(notneg) = pos$

The inverse functions compute on *sets*:

$\downarrow a = \{a' \in A \mid a' \sqsubseteq a\}$

$f^{\#-1}(S) = \{a \in A \mid f^{\#}(a) \in S\}$

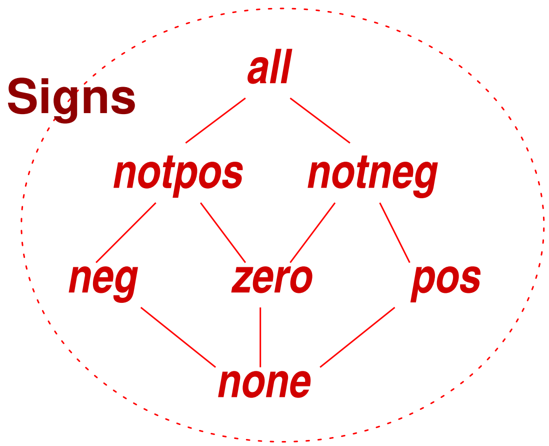
The entry condition can be used with a forwards analysis to generate postconditions that sharpen the assertions:

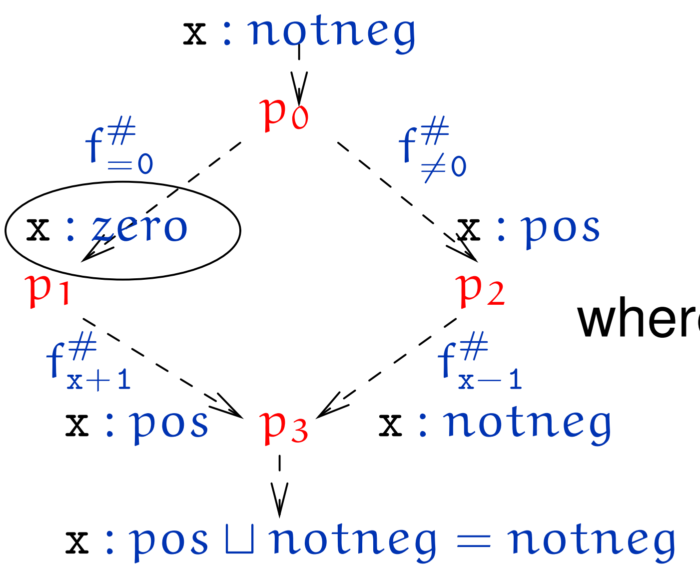
$\langle \mathtt{x} : notneg \rangle$ p_0: `if x=0`

p_1 : `x = x+1`

p_2 : `x = x-1`

p_3 : halt





where

$$f^\sharp_{=0}(a) = a \sqcap zero = \alpha \circ f_{=0} \circ \gamma$$

$$f^\sharp_{\neq 0} = \alpha \circ f_{\neq 0} \circ \gamma, \text{ e.g., } f^\sharp_{\neq 0}(notneg) = pos;$$

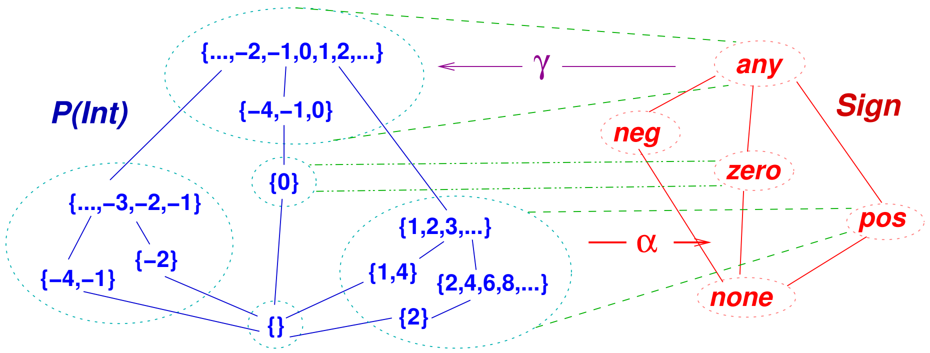
$$f^\sharp_{\neq 0}(zero) = \bot; f^\sharp_{\neq 0}(\top) = \top$$

$$f^\sharp_{+1} = \alpha \circ f_{+1} \circ \gamma, \text{ e.g., } f^\sharp_{+1}(notneg) = pos$$

The forwards-backwards analyses can be repeatedly alternated.

The “internal logic” of an abstract domain

Abstract values = logical propositions



Read properties like $neg \in Sign$ as logical propositions, "isNegative", etc.

For $S \subseteq C$, $a, a' \in A$, $\gamma : A \to PC$, define

- $S \models a$ iff $S \subseteq \gamma(a)$ e.g., $\{-3, -1\} \models neg$
- $a \vdash a'$ iff $a \sqsubseteq a'$ e.g., $neg \vdash any$

For $f : C \to PC$, $f^\sharp : A \to A$ is *sound* iff $f^* \circ \gamma \sqsubseteq \gamma \circ f^\sharp$ iff $\alpha \circ f^* \sqsubseteq f^\sharp \circ \alpha$ This makes $f^\sharp$ a *postcondition transformer*:

Proposition: $S \models a$ implies $f^*(S) \models f^\sharp(a)$.

$f^\sharp_{best} = \alpha \circ f^* \circ \gamma$ is the *strongest liberal postcondition* transformer for f.

A *has an* internal logic *that* γ *preserves*

First, treat all $a \in A$ as primitive propositions (*isNeg*, *isPos*, etc.).

A *has conjunction* when

$$S \models \phi_1 \sqcap \phi_2 \text{ iff } S \models \phi_1 \text{ and } S \models \phi_2, \text{ for all } S \subseteq C.$$

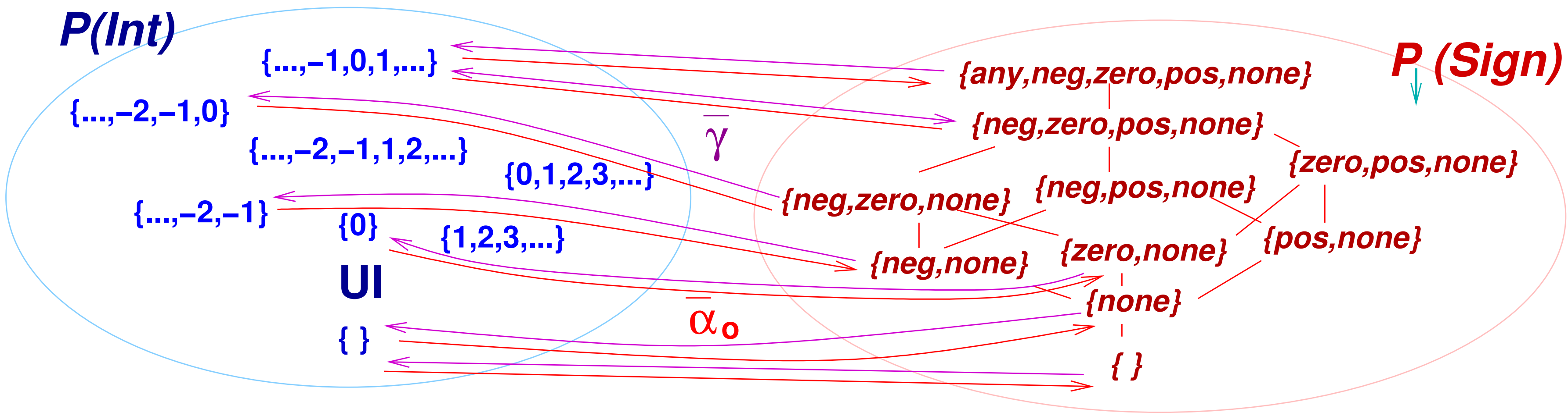
That is, $\gamma(\phi \sqcap \psi) = \gamma(\phi) \cap \gamma(\psi)$, for all $\phi, \psi \in A$.

Proposition: When $\gamma : A \to \mathrm{PC}$ is an upper adjoint, then A has conjunction.

Proposition: When $\gamma(\phi \sqcup \psi) = \gamma(\phi) \cup \gamma(\psi)$, then A has *disjunction*: $S \models \phi \sqcup \psi$ iff $S \models \phi$ or $S \models \psi$.

$Sign$ lacks disjunction: $\gamma(zero) \models neg \sqcup pos$, because $neg \sqcup pos = any$, but $\gamma(zero) \not\models neg$ and $\gamma(zero) \not\models pos$.

Sometimes, we can implement a domain's disjunctive completion [Cousots79,Giacobazzi00] :



$$(\mathcal{P}(\mathrm{int}), \subseteq)\langle\overline{\alpha_o}, \overline{\gamma}\rangle(\mathcal{P}_{\downarrow}(Sign), \subseteq)$$

$$\overline{\gamma}(T) = \cup_{a \in T}\, \gamma(a) \qquad \overline{\alpha_o}(S) = \downarrow\{\alpha\{c\} \mid c \in S\}$$

Downclosed sets are needed for monotonicity of key functions on the sets.

Now, $\overline{\gamma}$ preserves $\cap$ *and* $\cup$. Properties, $a \in A$, are interpreted in $\mathcal{P}_{\downarrow}(A)$ as $\overline{\alpha_o}(\gamma(a)) = \downarrow\{a\}$.

For $A = \mathcal{P}_{\downarrow}(Sign)$, these assertions are exact:

$$\phi \;::=\; neg \mid zero \mid pos \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2$$

Complete lattice A is *distributive* if $a \sqcap (b \sqcup c) = (a \sqcap b) \sqcup (a \sqcap c)$, for all $a, b, c \in A$. When $\sqcap$ is Scott-continuous, then

$$\phi \Rightarrow \psi \;\equiv\; \bigsqcup\{a \in A \mid a \sqcap \phi \sqsubseteq \psi\}$$

satisfies the property, $a \vdash \phi \Rightarrow \psi$ iff $a \sqcap \phi \vdash \psi$.

Proposition: If A is a distributive complete lattice, $\sqcap$ is Scott-continuous, and upper adjoint γ is 1-1, then A has *Heyting implication*, $\phi \Rightarrow \psi$, such that

$$S \models \phi \Rightarrow \psi \text{ iff } \gamma(\alpha(S)) \cap \gamma(\phi) \subseteq \gamma(\psi).$$

That is, $\gamma(\phi \Rightarrow \psi) = \bigcup\{S \in \gamma[A] \mid S \cap \gamma(\phi) \subseteq \gamma(\psi)\}$.

Heyting implication is weaker than classical implication, where $S \models \phi \Rightarrow \psi$ iff $S \cap \gamma(\phi) \subseteq \gamma(\psi)$ iff for all $c \in S$, if $\{c\} \models \phi$, then $\{c\} \models \psi$.

The POS domain for groundness analysis of logic programs uses Heyting implication [Cortesi91,Marriott93] .

If $\gamma(\bot_A) = \emptyset \in \mathcal{P}(\Sigma)$, we have falsity ($\bot$); this yields the logic,

$$\phi \ ::= \ a \mid \phi_1 \sqcap \phi_2 \mid \phi_1 \sqcup \phi_2 \mid \phi_1 \Rightarrow \phi_2 \mid \bot$$

In particular, $\neg\phi$ abbreviates $\phi \Rightarrow \bot$ and defines the *refutation* of ϕ within A, as done in the TVLA analyzer [Sagiv02] .

$\gamma : A \rightarrow PC$ is the interpretation function for the internal logic:

$$\gamma(a) = \text{given}$$

$$\gamma(\phi \sqcap \psi) = \gamma(\phi) \cap \gamma(\psi)$$

$$\gamma(\phi \sqcup \psi) = \gamma(\phi) \cup \gamma(\psi)$$

$$\gamma(\phi \Rightarrow \psi) = \bigcup\{S \in \gamma[A] \mid S \cap \gamma(\phi) \subseteq \gamma(\psi)\}$$

$$\gamma(\bot) = \emptyset$$

γ-completeness characterizes the internal logic

The interpretation for conjunction, $\gamma(\phi \sqcap \psi) = \gamma(\phi) \cap \gamma(\psi)$, shows that γ-completeness is *exactly* the criterion for determining the connectives in A's internal logic:

Proposition: For $f : C^n \to PC$, A's logic includes connective $f^\sharp$ iff $f^\sharp$ is γ-complete for f^*:

$$\gamma(f^\sharp(\phi_1, \phi_2, \cdots)) = f^*(\gamma(\phi_1), \gamma(\phi_2), \cdots)$$

Example: For $Sign = \{none, neg, zero, pos, any\}$, $\mathtt{negate}^\sharp$ is γ-complete for $\mathtt{negate}(S) = \{-n \mid n \in S\}$ (where $\mathtt{negate}^\sharp(pos) = neg$, $\mathtt{negate}^\sharp(neg) = pos$, $\mathtt{negate}^\sharp(zero) = zero$, etc.):

$$\phi \ ::= \ a \mid \phi_1 \sqcap \phi_2 \mid \mathtt{negate}^\sharp(\phi)$$

We can state "negate" assertions, e.g., $pos \models \mathtt{negate}^\sharp(neg \sqcap any)$.

Post-image (left-to-right) abstraction of relations

$f : C \to PC$ defines a relation in $C \times C$, e.g., $\{1, 3\}[\![succ]\!]\{2, 4\}$.

f's left-to-right (post) image, $post_f : PC \to PC$, is

$$post_f(S) = \cup_{c \in S} f(c).$$

For Galois connection, $PC\langle\overline{\alpha_o}, \overline{\gamma}\rangle\mathcal{P}_\downarrow(A)$, and $f^\sharp : A \to \mathcal{P}_\downarrow(A)$,

- for $T \in \mathcal{P}_\downarrow(A)$, define $post_{f^\sharp}(T) = \sqcup_{a \in T} f^\sharp(a) = \cup_{a \in T} f^\sharp(a)$.
- use $post_{f^\sharp}$ to compute left-to-right (over)approximations of f, e.g., $\{neg\}[\![succ^\sharp]\!]\{neg, zero\}$, that is, $neg \vee zero$.

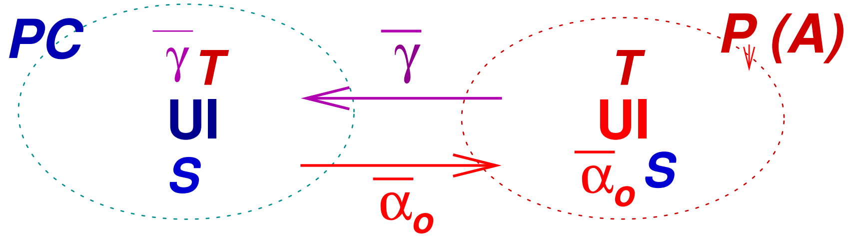
Proposition: For $f^\sharp_{best} = \overline{\alpha_o} \circ f^* \circ \gamma$,

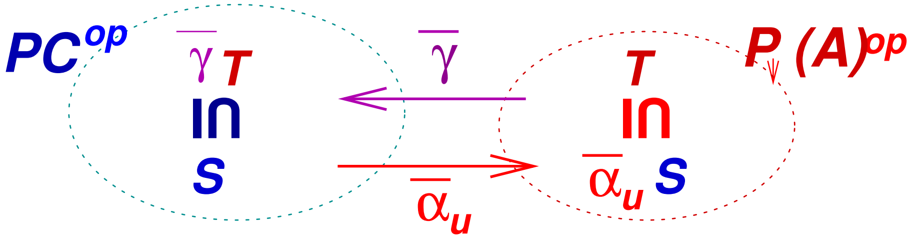
$$(post_f)^\sharp_{best} = \overline{\alpha_o} \circ post_f \circ \overline{\gamma} = post_{f^\sharp_{best}}$$

Corollary: If f is γ-complete, then $(post_{f^\sharp_{best}}\ \phi)$ is in $\mathcal{P}_\downarrow(A)$'s logic.

Given $PC\langle\alpha,\gamma\rangle A$, *we have* **two** *relevant Galois connections between* PC *and* $\mathcal{P}_{\downarrow}(A)$

Recall that $\overline{\gamma}(T) = \cup_{a \in T}\, \gamma(a)$ and that $\overline{\gamma}$ preserves both unions and intersections on $\mathcal{P}_{\downarrow}(A)$. Therefore, $\overline{\gamma}$ is an upper adjoint in *two* different ways:





Overapproximating abstraction:

$$\begin{aligned}\overline{\alpha_o}(S) &= \bigcap\{T \mid S \subseteq \overline{\gamma}(T)\}\\ &= \downarrow\!\{\alpha\{c\} \mid c \in S\}\end{aligned}$$

where

$\downarrow T = \{a \mid$ exists $a' \in T, a \sqsubseteq a'\}$.

Underapproximating abstraction:

$$\begin{aligned}\overline{\alpha_u}(S) &= \bigcup\{T \mid \overline{\gamma}(T) \subseteq S\}\\ &= \{a \mid \gamma(a) \subseteq S\}\end{aligned}$$

where

$(D, \sqsubseteq_D)^{op}$ is $(D, \sqsupseteq_D)$.

Pre-image (right-to-left) abstraction of relations

$f : C \rightarrow PC$ defines a relation $\subseteq C \times C$, e.g., $\{0, 1, 3\}[\![succ]\!]\{1, 2, 4\}$.

f's right-to-left (pre) image, $\widetilde{pre}_f : PC \rightarrow PC$, is

$$\widetilde{pre}_f(S) = \cup\{S' \subseteq C \mid f^*(S') \subseteq S\} = \{c \mid f(c) \subseteq S\}$$

For Galois connection, $PC^{op}\langle\overline{\alpha_u}, \overline{\gamma}\rangle\mathcal{P}_\downarrow(A)^{op}$ and $f^\sharp : A \rightarrow \mathcal{P}_\downarrow(A)$,

- for $T \in \mathcal{P}_\downarrow(A)$, define $\widetilde{pre}_{f^\sharp} = \{a \mid f^\sharp(a) \subseteq T\}$
- use $\widetilde{pre}_{f^\sharp}$ to compute right-to-left (under)approximations of f, e.g., $zero \vee pos[\![succ^\sharp]\!]pos$ and $none[\![succ^\sharp]\!]zero$ (!)

Theorem: $(\widetilde{pre}_f)^\sharp_{best} = \overline{\alpha_u} \circ \widetilde{pre}_f \circ \overline{\gamma} = \widetilde{pre}_{f^\sharp_{best}}$.

Because $\widetilde{pre}_{f^\sharp}\phi$ always underapproximates $\widetilde{pre}_f(\overline{\gamma}(\phi))$, it can be added to $\mathcal{P}_\downarrow(A)$'s logic.

Indeed, we can always define an underapproximating **external logic**

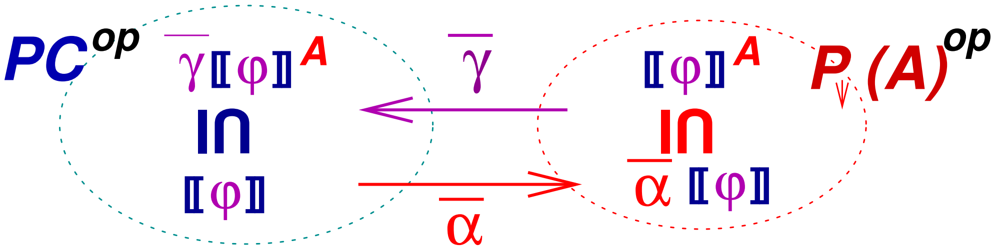
For each concrete property of interest, $[\![\phi]\!] \subseteq C$, define

$$[\![\phi]\!]^{\mathcal{A}} = \{a \in A \mid \gamma(a) \subseteq [\![\phi]\!]\}$$

Then, assert $a \vdash \phi$ iff $a \in [\![\phi]\!]^{\mathcal{A}}$.

This definition follows from the underapproximating Galois connection:

$$\overline{\gamma}(T) = \bigcup\{\gamma(a) \mid a \in T\}$$
$$\overline{\alpha_u}(S) = \{a \mid \gamma(a) \subseteq S\}$$



That is, $[\![\phi]\!]^{A} = \overline{\alpha_u}[\![\phi]\!]$.

The inverted ordering gives *underapproximation*: $[\![\phi]\!] \supseteq \overline{\gamma}([\![\phi]\!]^{\mathcal{A}})$. This form of external logic is standard in "abstract model checking."

The inductively defined underapproximation to $\overline{\alpha_u}[\![\phi]\!]$:

$$[\![a]\!]^{\mathcal{A}}_{\mathrm{ind}} = \overline{\alpha_u}(\gamma(a))$$
$$[\![\phi_1 \wedge \phi_2]\!]^{\mathcal{A}}_{\mathrm{ind}} = [\![\phi_1]\!]^{\mathcal{A}}_{\mathrm{ind}} \cap [\![\phi_2]\!]^{\mathcal{A}}_{\mathrm{ind}}$$
$$[\![\phi_1 \vee \phi_2]\!]^{\mathcal{A}}_{\mathrm{ind}} = [\![\phi_1]\!]^{\mathcal{A}}_{\mathrm{ind}} \cup [\![\phi_2]\!]^{\mathcal{A}}_{\mathrm{ind}}$$
$$[\![[f]\phi]\!]^{\mathcal{A}}_{\mathrm{ind}} = \widetilde{pre}_{f^\sharp}[\![\phi]\!]^{\mathcal{A}}_{\mathrm{ind}} = \{a \in A \mid f^\sharp(a) \in [\![\phi]\!]^{\mathcal{A}}_{\mathrm{ind}}\}$$

Entailment and provability are as expected: $a \models \phi$ iff $\gamma(a) \subseteq [\![\phi]\!]$, and $a \vdash \phi$ iff $a \in [\![\phi]\!]^{\mathcal{A}}_{\mathrm{ind}}$.

Soundness ($\vdash$ implies $\models$) is immediate, and completeness ($\models$ implies $\vdash$) follows when $\overline{\alpha_u} \circ [\![\cdot]\!] = [\![\cdot]\!]^{\mathcal{A}}_{\mathrm{ind}}$. This is called *logical best preservation* or *logical $\overline{\alpha}$-completeness* [Cousots00,Schmidt06] .

Scaling upwards

Analyzing large (100K+ LOC) programs

- engineered as a *one-pass analysis*, like static data-type checking
- *flow-insensitive* (ignores control-test expressions, loop iterations, distinct procedure-call points).
- *"whole-program analysis"*: examines entire source-code base

The standard example is pointer analysis on C programs, where properties are stated, *"var* x *may-point-to vars* $\{y, z, ...\}$*."* A set of equations are generated in one program pass and solved in some small bound of iterations [Andersen94, Steensgaard96, HeintzeTardieu04] .

Advantages: simple, fast, complete code coverage, no hand-extracted "abstract model" (as required for model-checking) [Engler04]

Drawbacks: properties are simple, too many "false alarms" (inability to verify desired property)

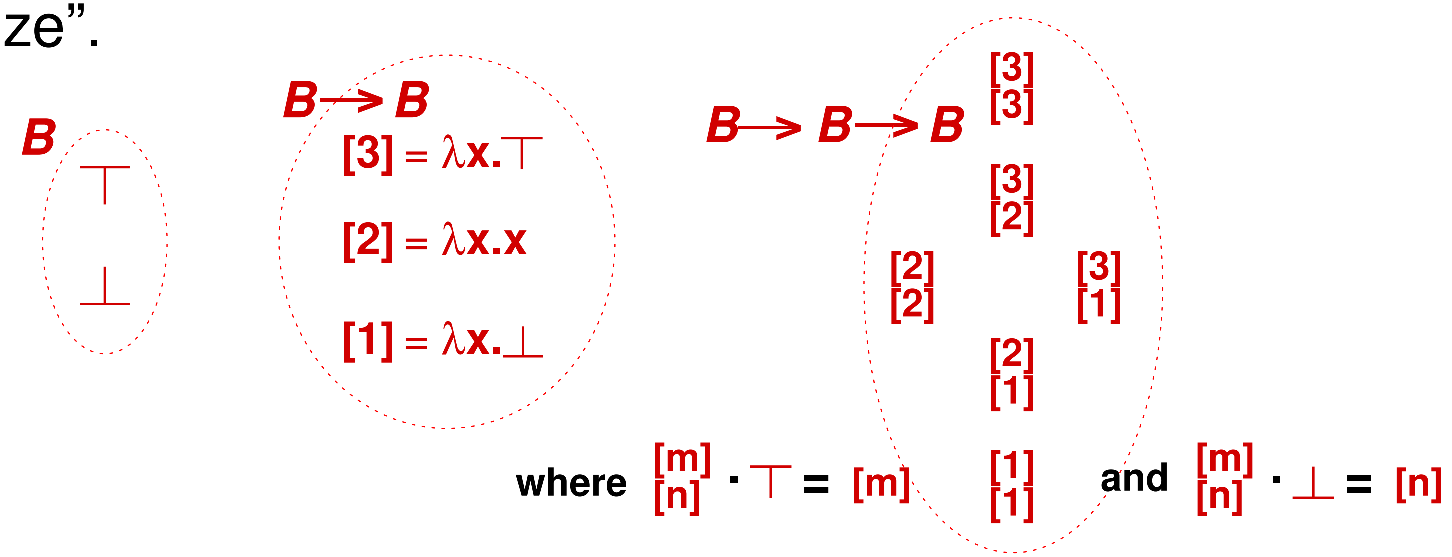
Modular analysis

- A program unit is abstracted and analyzed to a *summary structure* or *assume-guarantee relation*, where properties of the unit's free variables/inputs are associated/mapped to properties of the unit/outputs.
- When units are linked, so are their summaries, generating a composite summary. *We don't reanalyze the units.*
- Practical (better than linear-time) speedups are obtained when fixed points are solved locally within each unit (*and not at link time*) [CousotCousot02] .

There is no ideal approach, especially for the last item, so we survey some techniques (*summaries, frontiers, symbolic evaluation*) using the classic example of abstracting a higher-order function definition.

Example: higher-order normalization ("strictness") analysis

$B = \{\perp, \top\}$, where $\top$ means "might normalize" and $\perp$ means "does not normalize".



Example: `F m n = if (m=0) (n) (F (m+1) n)`

$F^\sharp = \lambda a\colon B.\lambda b\colon B.\ a \sqcap (b \sqcup (F^\sharp\ a\ b))$

$\mathrm{graph}(F^\sharp) = \{\perp \mapsto \perp \mapsto \perp,\ \perp \mapsto \top \mapsto \perp,\ \top \mapsto \perp \mapsto \perp,\ \top \mapsto \top \mapsto \top\}$.

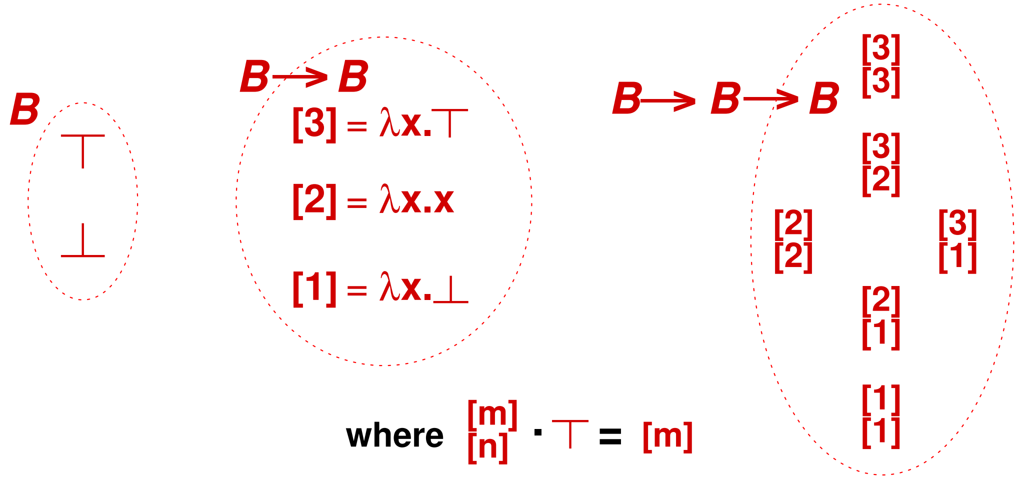
Domain B can be applied to analyses that predict the outcome of a boolean predicate/invariant ("predicate abstraction").

A higher-order, module-like example

Define: $F^\sharp = \lambda f : B \to (B \to B).\lambda x : B.\ (x, f \cdot x)$

The function's graph (summary table) has 12 entries:

$$
\begin{aligned}
&\mathrm{graph}(F^\sharp) = \{ \\
&\left[\begin{smallmatrix}1\\1\end{smallmatrix}\right] \mapsto \bot \mapsto (\bot, \bot), \\
&\left[\begin{smallmatrix}1\\1\end{smallmatrix}\right] \mapsto \top \mapsto (\top, \bot), \\
&\left[\begin{smallmatrix}2\\1\end{smallmatrix}\right] \mapsto \bot \mapsto (\bot, \bot), \\
&\left[\begin{smallmatrix}2\\1\end{smallmatrix}\right] \mapsto \top \mapsto (\top, \top), \\
&\quad \cdots \\
&\left[\begin{smallmatrix}3\\3\end{smallmatrix}\right] \mapsto \bot \mapsto (\bot, \top), \\
&\left[\begin{smallmatrix}3\\3\end{smallmatrix}\right] \mapsto \top \mapsto (\top, \top) \ \}
\end{aligned}
$$



where $\left[\begin{smallmatrix}m\\n\end{smallmatrix}\right] \cdot \top = [m]$

and $\left[\begin{smallmatrix}m\\n\end{smallmatrix}\right] \cdot \bot = [n]$

It's model-checking-like
and feasible to implement!

Partial summary/graph: **frontier** *[Clack&PeytonJones85]*

Assemble the graph in increments and retain only useful ("frontier") entries, as based on these consequences of monotonicity:

- if $a \mapsto b \in \mathrm{frontier}(F^\sharp)$, then *(i)* for all $a' \sqsubseteq a$, $a' \mapsto b$ is sound; *(ii)* for all $b' \sqsubseteq b$, $a \mapsto b'$ is sound.
- if $a \mapsto \top \in \mathrm{frontier}(F^\sharp)$, then forall $a' \sqsupseteq a$, $a' \mapsto \top$ is sound.
- if $a_1 \mapsto b_1, a_2 \mapsto b_2 \in \mathrm{frontier}(F^\sharp)$, then *(i)* $a_1 \sqcap a_2 \mapsto b_1 \sqcap b_2$ is sound; *(ii)* if $F^\sharp$ preserves $\sqcup$ (holds when $F^\sharp$'s domain is a disjunctive completion), then $a_1 \sqcup a_2 \mapsto b_1 \sqcup b_2$ is sound.

Example frontier: for $F^\sharp = \lambda f : B \rightarrow (B \rightarrow B).\lambda x : B.\ (x, f \cdot x)$,

$$\mathrm{frontier}(F^\sharp) = \left\{ \begin{array}{l} [\begin{smallmatrix}2\\2\end{smallmatrix}] \mapsto \bot \mapsto (\bot, [2]),\ [\begin{smallmatrix}2\\2\end{smallmatrix}] \mapsto \top \mapsto (\top, [2]), \\ [\begin{smallmatrix}3\\1\end{smallmatrix}] \mapsto \top \mapsto (\top, [3]) \end{array} \right\}$$

Example inferences based on the frontier

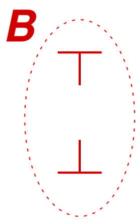
For $F^\sharp = \lambda f : B \to (B \to B).\lambda x : B.\,(x, f \cdot x)$,

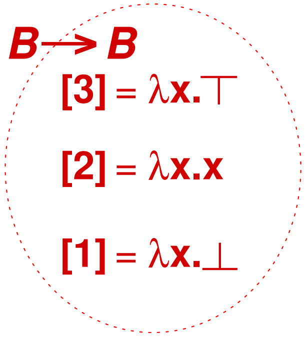
$\mathrm{frontier}(F^\sharp) = \{$

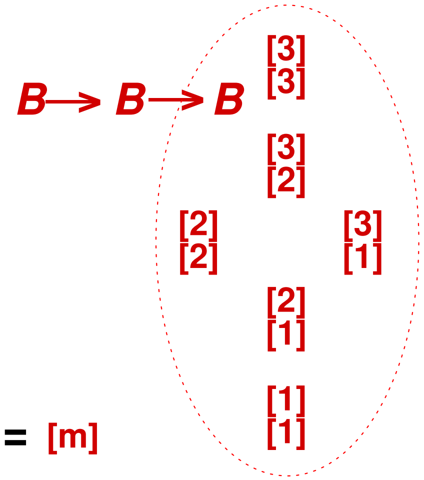
$\left[\begin{smallmatrix}2\\2\end{smallmatrix}\right] \mapsto \bot \mapsto (\bot, [2])$,

$\left[\begin{smallmatrix}2\\2\end{smallmatrix}\right] \mapsto \top \mapsto (\top, [2])$,

$\left[\begin{smallmatrix}3\\1\end{smallmatrix}\right] \mapsto \top \mapsto (\top, [3])\ \}$,







where $\left[\begin{smallmatrix}m\\n\end{smallmatrix}\right] \cdot \top = [m]$

and $\left[\begin{smallmatrix}m\\n\end{smallmatrix}\right] \cdot \bot = [n]$

we can conclude

$\left[\begin{smallmatrix}2\\1\end{smallmatrix}\right] \mapsto \top \mapsto (\top, [3])$ is sound (because $\left[\begin{smallmatrix}2\\1\end{smallmatrix}\right] \sqsubseteq \left[\begin{smallmatrix}3\\1\end{smallmatrix}\right]$)

$\left[\begin{smallmatrix}3\\3\end{smallmatrix}\right] \mapsto \top \mapsto (\top, [3])$ is sound (because $\left[\begin{smallmatrix}3\\1\end{smallmatrix}\right]$, $\top$ map to $(\top, [3])$)

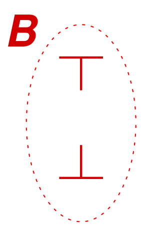
$\left[\begin{smallmatrix}2\\1\end{smallmatrix}\right] \mapsto \top \mapsto (\top, [2])$ is sound (because $\left[\begin{smallmatrix}2\\1\end{smallmatrix}\right] = \left[\begin{smallmatrix}2\\2\end{smallmatrix}\right] \sqcap \left[\begin{smallmatrix}3\\1\end{smallmatrix}\right]$)

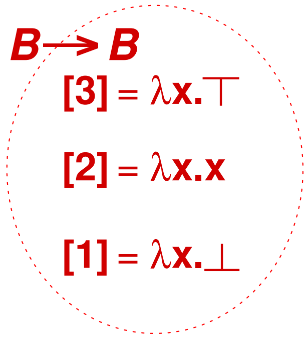
$\left[\begin{smallmatrix}3\\2\end{smallmatrix}\right] \mapsto \top \mapsto (\top, [3])$ is sound (because $\left[\begin{smallmatrix}3\\2\end{smallmatrix}\right] = \left[\begin{smallmatrix}2\\2\end{smallmatrix}\right] \sqcup \left[\begin{smallmatrix}3\\1\end{smallmatrix}\right]$)

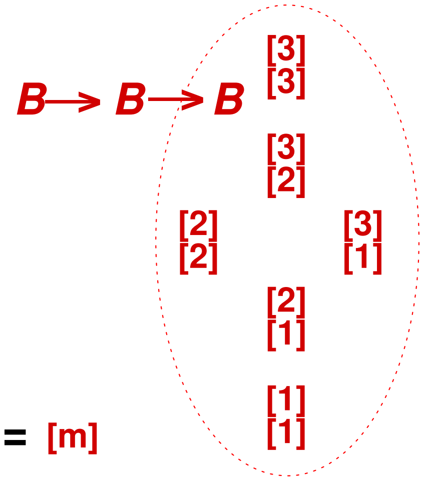
Integrating symbolic evaluation with frontiers

For $F^\sharp = \lambda f : B \to (B \to B).\lambda x : B.\ (x, f \cdot x)$,

$$\begin{aligned}
&\mathrm{symbolicFrontier}(F^\sharp) = \{ \\
&\quad \left[{2 \atop 2}\right] \mapsto a \mapsto (a, a), \\
&\quad \left[{3 \atop 1}\right] \mapsto a \mapsto (a, \left[{3 \atop 1}\right] \cdot a), \\
&\quad f \mapsto a \mapsto (a, f \cdot a) \quad \}
\end{aligned}$$







where $\left[{m \atop n}\right] \cdot \top = [m]$

and $\left[{m \atop n}\right] \cdot \bot = [n]$

- Starting from a purely symbolic formulation (the third line), the frontier expands with useful instances.
- At any point, we can replace symbolic arguments by $\top$ to "close" the frontier, generating a "worst case analysis."
- We can apply algebraic techniques to solve local fixed points.

Solving local fixed points (intuition)

Example: `F x = if (...)  (g x) (h(F(f x))`

$$F^\sharp = \bigsqcup_{i\geq 0} F_i, \quad \text{where} \quad \begin{aligned} F_0 &= \lambda a.\bot \\ F_{i+1} &= \lambda a.(g\ a) \sqcup (h(F_i(f\ a))) \end{aligned}$$

By inductive reasoning,

$$\begin{aligned} F_i &= \bigsqcup_{0\leq j<i} h^j(g(f^j\ a)) \\ &\sqsubseteq \bigsqcup_{0\leq j<i} h^j(g(f^*\ a)) \\ &\sqsubseteq h^*(g(f^*\ a)) \end{aligned} \quad \text{where} \quad \begin{aligned} f^i &= f \circ f \circ \cdots \circ f, i \text{ times} \\ f^* &= \bigsqcup_{j\geq 0} f^j \end{aligned}$$

Each occurrence of f^* is solved locally, cheaply. The reasoning is implemented with regular tree/expression techniques; precision is traded for speed-up [CousotCousot02,Moeller03] .

References

1. *This talk:* `santos.cis.ksu.edu/schmidt/SLS13.pdf`
2. B. Blanchet, P. Cousot, et al. A static analyzer for large safety-critical software. ACM PLDI 2003.
3. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs. ACM POPL 1977.
4. Patrick Cousot, Radhia Cousot: Modular Static Program Analysis. CC 2002, Springer LNCS 2304.
5. F. Nielson, H.R. Nielson, C. Hankin. *Principles of Program Analysis.* Springer 2010.
6. D. Schmidt. Trace-Based Abstract Interpretation of Operational Semantics. J. Lisp and Symbolic Computation 10 (1998).
7. D. Schmidt. Internal and External Logics of Abstract Interpretations. VMCAI 2008, Springer LNCS 4905.