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 ($\cap$). See example, $\text{Sign}$, 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 $a$.”

$\gamma$ must preserve meets – for $T \subseteq A$, $\gamma(\cap T) = \bigcap_{a \in T} \gamma(a)$ – so that an inverse function, $\alpha: PC \rightarrow A$, can be defined.
Operations \( f \) are abstracted to \( f^\# \) to compute on \( A \)

\[
\begin{align*}
\text{readInt}(x) & \quad \text{Q: is the output pos?} \\
x = \text{succ}(x) & \quad \text{A: abstractly interpret} \\
\text{if } x < 0 : & \quad \text{input domain Int by} \\
x = \text{negate}(x) & \quad \text{Sign} = \\
\text{else:} & \quad \{ \text{neg, zero, pos, any, none}\}: \\
x = \text{succ}(x) & \\
\text{writeInt}(x) & \quad \text{readSign}(x) \\
\text{writeInt}(x) & \quad \text{writeSign}(x)
\end{align*}
\]

\[
\begin{align*}
\text{succ}^\#(pos) = pos & \quad \text{negate}^\#(neg) = pos \\
\text{succ}^\#(zero) = pos & \quad \text{negate}^\#(zero) = zero \\
\text{succ}^\#(neg) = \text{any} (!) & \quad \text{negate}^\#(pos) = neg \\
\text{succ}^\#(any) = \text{any} & \quad \text{negate}^\#(any) = \text{any} \\
\text{where} & \quad \text{and}
\end{align*}
\]

For the abstract data-test sets, zero, neg, pos, we calculate:

\[
\{ \text{zero} \mapsto \text{pos}, \text{pos} \mapsto \text{pos}, \text{neg} \mapsto \text{any} \}. \text{ The last result arises because}
\]

\[
\text{succ}^\#(\text{neg}) = \text{any} \quad \text{and} \quad \text{filterNeg}(\text{any}) = \text{neg} \quad \text{(good!)} \quad \text{but} \quad \text{filterNonNeg}(\text{any}) = \text{any} \quad \text{(bad — we need zero \lor pos!), so we cannot ensure the success of the else-arm.}
\]
A Galois connection formalizes the abstraction

\[ \gamma : \text{Sign} \rightarrow \mathcal{P}(\text{Int}) \]

\[ \gamma(\text{none}) = \{\} \]
\[ \gamma(\text{any}) = \text{Int} \]
\[ \gamma(\text{neg}) = \{\ldots, -3, -2, -1\} \]
\[ \gamma(\text{zero}) = \{0\} \]
\[ \gamma(\text{pos}) = \{1, 2, 3, \ldots\} \]

\[ \alpha : \mathcal{P}(\text{Int}) \rightarrow \text{Sign} \]

\[ \alpha(S) = \bigcap \{a | \gamma(a) \subseteq S\} \]

\[ \text{e.g., } \alpha\{2, 4, 6, 8, \ldots\} = \text{pos}, \]
\[ \alpha\{-1, 0\} = \text{any}, \alpha\{0\} = \text{zero} \]

\[ (\mathcal{P}(\text{Int}), \subseteq) \langle \alpha, \gamma \rangle (\text{Sign}, \sqsubseteq) \text{ is a Galois connection:} \]

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

\[ \gamma \text{ interprets the elements in Sign, and } \alpha \text{ maps each data-test set in } \mathcal{P}(\text{Int}) \text{ to the property that best describes the set [CousotCousot77].} \]
An abstract operation is monotone and sound

\( f^\# : A \rightarrow A \) is \textit{sound} for \( f : C \rightarrow PC \) iff \( \alpha \circ f^* \subseteq f^\# \circ \alpha \)

(iff \( f^* \circ \gamma \subseteq \gamma \circ f^\# \)):

\[
\begin{align*}
\alpha \downarrow & \quad f^* \downarrow \alpha & \gamma(\ a) & \xrightarrow{f^*} \bullet \\
\alpha(\ S) & \xrightarrow{f^\#} \prod \quad \gamma(\ a) & \xrightarrow{\gamma \circ f^\#} \gamma(\ f^\#(a)) \quad \alpha \text{ and } \gamma \text{ act as semi-homomorphisms.}
\end{align*}
\]

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

\( f^\# \) is a \textit{postcondition transformer}: \( S \subseteq \gamma(\ a) \) implies
\( f^*(S) \subseteq \gamma(\ f^\#(a)) \) where \( f^*(S) = \bigcup_{c \in S} f(c) \).

\( f^\#_{\text{best}} = \alpha \circ f^* \circ \gamma \) is the \textit{strongest (liberal) postcondition} transformer.

**Definition:** \( f^\# \) is \( \gamma \)-\textit{complete} ("forwards complete") for \( f \) iff
\( f^* \circ \gamma = \gamma \circ f^\# \) [Giacobazzi01]. \( f^\# \) is \( \alpha \)-\textit{complete} ("backwards complete") for \( f \) iff \( \alpha \circ f^* = f^\# \circ \alpha \) [Cousots00].
An aggregate, e.g., $\text{Var} \rightarrow \mathbb{C}$, can be abstracted pointwise or relationally.

**Sign:** $[x \mapsto \geq 0][y \mapsto \geq 0]$

**Interval:** $[x \mapsto [3, 27]][y \mapsto [4, 32]]$

Octagon: $\bigwedge_i (\pm x_i \pm y_i \leq c_i)$

Polyhedron: $\bigwedge_i (\sum_j a_{ij} \cdot x_{ij} \leq b_i)$

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

\[
\begin{align*}
(a) \quad & \begin{cases}
V_2 - V_1 \leq 4 \\
V_1 - V_2 \leq -1 \\
V_3 - V_1 \leq 3 \\
V_1 - V_3 \leq -1 \\
V_2 - V_3 \leq 1
\end{cases} \\
(b) \quad & \begin{bmatrix}
1 & 2 & 3 \\
1 & +\infty & 4 & 3 \\
2 & -1 & +\infty & +\infty \\
3 & -1 & 1 & +\infty
\end{bmatrix}
\end{align*}
\]

\(\gamma_{pot}(m)\) (d).

Figure 2. A potential constraint conjunction (a), its corresponding DBM \(m\) (b), potential graph \(G(m)\) (c), and potential set concretization \(\gamma_{pot}(m)\) (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 $z \geq x \land z \geq y$ at $p_3$:

$p_0$: if $x < y$

$p_1$: then $z = y$

$p_2$: else $z = x$

$p_3$: exit

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

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

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

At all occurrences of $p_3$ in the abstract trace, $\phi_2 \land \phi_3$ holds.
When a goal is undecided, domain refinement becomes necessary

Prove $\phi_0 \equiv x \geq y$ at $p_4$:

$p_0 : \text{if } !(x \geq y)$
$p_1 : \text{then } \{ i = x; \}$
$p_2 : x = y;$
$p_3 : y = i;$
$p_4 : \} $

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

because $x \not\geq y$ and $x \geq i$

imply $y > i$ implies $x_{\text{new}} \geq i$
But incremental predicate refinement cannot synthesize many interesting loop invariants. For this example:

\[ p_0 : \ i = n; \ x = 0; \]

\[ p_1 : \ \textbf{while} \ i \neq 0 \ \{ \]
\[ p_2 : \ x = x + 1; \ i = i - 1; \]
\[ \} \]

\[ p_3 : \ \textbf{goal:} \ x = n \]

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!

\textit{The loop invariant is} \( x = n - i \quad :-) \)
Mechanics of static analysis: abstracting small-step and big-step semantics definitions
The most basic static analysis is trace generation

\[ \begin{align*}
 p_0 & : \text{ while } (x \neq 1) \{ \\
 p_1 & : \text{ if } \text{Even}(x) \\
 p_2 & : \text{ then } x = x \div 2; \\
 p_3 & : \text{ else } x = 3 \times x + 1; \\
 p_4 & : \text{ exit}
\end{align*} \]

Two concrete traces:

\[ \begin{align*}
 p_0,4 & \downarrow \\
 p_1,4 & \downarrow \\
 p_2,4 & \downarrow \\
 p_0,2 & \downarrow \\
 p_1,2 & \downarrow \\
 p_2,2 & \downarrow \\
 p_0,1 & \downarrow \\
 p_4,1 & \downarrow \\
 p_0,6 & \downarrow \\
 p_1,6 & \downarrow \\
 p_2,6 & \downarrow \\
 p_0,3 & \downarrow \\
 p_1,3 & \downarrow \\
 p_2,3 & \downarrow \\
 p_0,10 & \downarrow \\
 \vdots & \downarrow \\
 p_0,1 & \downarrow \\
 p_4,1 & \downarrow
\end{align*} \]

For \( Parity = \{\text{none, even, odd, any}\} \), the loop’s operations, \( f \), are abstracted to \( f^\# \). 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, $\text{ProgramPoint} \rightarrow \mathbb{A}$

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

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

Flow equations for previous example:

- $\text{init} = \langle x: \text{even} \rangle$
- $p_0\text{Store} = \text{init} \sqcup f_2^\#(p_2\text{Store}) \sqcup f_3^\#(p_3\text{Store})$
- $p_1\text{Store} = f_{0\text{t}}^\#(p_0\text{Store})$
- $p_2\text{Store} = f_{1\text{t}}^\#(p_1\text{Store})$
- $p_3\text{Store} = f_{1\text{f}}^\#(p_1\text{Store})$
- $p_4\text{Store} = f_{0\text{f}}^\#(p_0\text{Store})$
We *solve* the flow equations by calculating approximate solutions in stages until *the least fixed point* is reached.

**Note:** \( \bot \) is the same as \( \langle x: \bot \rangle \)

<table>
<thead>
<tr>
<th>stage</th>
<th>( p_0 \text{Store} )</th>
<th>( p_1 \text{Store} )</th>
<th>( p_2 \text{Store} )</th>
<th>( p_3 \text{Store} )</th>
<th>( p_4 \text{Store} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( \bot )</td>
<td>( \bot )</td>
<td>( \bot )</td>
<td>( \bot )</td>
<td>( \bot )</td>
</tr>
<tr>
<td>1</td>
<td>( \langle x: \text{even} \rangle )</td>
<td>( \bot )</td>
<td>( \bot )</td>
<td>( \bot )</td>
<td>( \bot )</td>
</tr>
<tr>
<td>2</td>
<td>( \langle x: \text{even} \rangle )</td>
<td>( \langle x: \text{even} \rangle )</td>
<td>( \bot )</td>
<td>( \bot )</td>
<td>( \bot )</td>
</tr>
<tr>
<td>3</td>
<td>( \langle x: \text{even} \rangle )</td>
<td>( \langle x: \text{even} \rangle )</td>
<td>( \langle x: \text{even} \rangle )</td>
<td>( \bot )</td>
<td>( \bot )</td>
</tr>
<tr>
<td>4</td>
<td>( \langle x: \text{any} \rangle )</td>
<td>( \langle x: \text{even} \rangle )</td>
<td>( \langle x: \text{even} \rangle )</td>
<td>( \bot )</td>
<td>( \bot )</td>
</tr>
<tr>
<td>5</td>
<td>( \langle x: \text{any} \rangle )</td>
<td>( \langle x: \text{any} \rangle )</td>
<td>( \langle x: \text{even} \rangle )</td>
<td>( \bot )</td>
<td>( \langle x: \text{odd} \rangle )</td>
</tr>
<tr>
<td>6</td>
<td>( \langle x: \text{any} \rangle )</td>
<td>( \langle x: \text{any} \rangle )</td>
<td>( \langle x: \text{even} \rangle )</td>
<td>( \langle x: \text{odd} \rangle )</td>
<td>( \langle x: \text{odd} \rangle )</td>
</tr>
</tbody>
</table>

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*,

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

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

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

This example terminates: \(i\)'s ranges are

- at \(p_1\) : [0..9]
- at \(p_2\) : [1..10]
- at loop exit : [1..10] \(\sqcup\) [10, +\(\infty\)] = [10, 10]
But others might not, because the domain is not finite height:

\[ i = 0; \quad \square \quad i = [0,0] \]
\[ \text{while true } \{ \quad \square \quad i = [0,0] \quad \bigcup \quad [1,1] \quad \bigcup \quad [2, 2] \ldots \]
\[ i = i + 1; \quad \square \quad \text{infinite limit is } [0, + \infty] \]
\[ \square \quad i = [ ] \quad \text{(dead code)} \]

The analysis generates the infinite sequence of stages, 
\([0, 0], [0, 1], \ldots, [0, i], \ldots\) 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 \(\triangledown\), called a *widening operator*:

\[
[ ] \triangledown [i, j] = [i, j] \quad [i, j] \triangledown [i', j'] = \begin{cases} 
\text{if } i' < i \text{ then } -\infty \text{ else } i, \\
\text{if } j' > j \text{ then } +\infty \text{ else } j 
\end{cases}
\]
The widening operator, which guarantees finite convergence for all increasing sequences on the interval domain, quickly terminates the example:

```plaintext
i = 0;  \[\leq\] \quad i = [0,0] 
while true {
    \ldots  \[\leq\] \quad i = [0,0] \triangledown [1,1] = [0, +\infty] 
i = i + 1;
}  \[\leq\] \quad i = [ ]  \quad \text{(dead code)}
```

but in general, it can lose much precision:

```plaintext
int a = new int[10];  
i = 0;  \[\leq\] \quad i = [0,0] 
while (i < 10) {  
    \ldots a[i] \[\leq\] \quad i = [0,0] \triangledown [1,1] = [0, +\infty] 
i = i + 1;
}  \[\leq\] \quad i = [10, +\infty]
```
For this reason, a complementary operation, \( \triangle \), called a *narrowing operation*, can be used after \( \nabla \) 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 $B$:

$$C ::= \ p : x = E | C | if \ E \ C_1 \ C_2 | while \ E \ C$$

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

$$[p : x = E]_{in} = f^p_{p}(in) \quad \text{(the transfer function for } p)$$

$$[C]_{in} = [C_2]([C_1]_{in})$$

$$[if \ E \ C_1 \ C_2]_{in} = [C_1]_{in} \sqcup [C_2]_{in}$$

$$[while \ E \ C]_{in} = \text{in} \sqcup \text{out}_C,$$

where $\text{out}_C = \bigsqcup_{i \geq 0} \text{out}_i$,

and $\text{out}_0 = \bot_A$ and $\text{out}_{i+1} = [C](\text{in} \sqcup \text{out}_i)$
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:

\[
[ p : x = E ] \text{in} = \text{in} - \text{kill}_x \cup \{ p \} \\
[C] \text{in} = [[C_2]]([C_1] \text{in}) \\
[\text{if } E \ C_1 C_2] \text{in} = [C_1] \text{in} \cup [C_2] \text{in} \\
[\text{while } E \ C] \text{in} = \text{in} \cup \bigcup_{i \geq 0} \text{out}_i, \\
\text{where } \text{out}_0 = \{ \} \\
\text{and } \text{out}_{i+1} = [C](\text{in} \cup \text{out}_i)
\]
Big-step relational semantics: derivation trees

\[ \sigma \vdash p : x = E \downarrow f_p(\sigma) \]

\[ \begin{align*}
\sigma \vdash C_1 \downarrow \sigma_1 & \quad \sigma_1 \vdash C_2 \downarrow \sigma_2 \\
\sigma \vdash C_1 ; C_2 \downarrow \sigma_2 & \quad f_{E_t}(\sigma) \vdash C_1 \downarrow \sigma_1 & f_{E_f}(\sigma) \vdash C_2 \downarrow \sigma_2 \\
\end{align*} \]

\[ \sigma \vdash \text{if } E \ C \downarrow \sigma'' \quad \sigma' \vdash \text{while } E \ C \downarrow \sigma'' \quad \sigma' \vdash \text{while } E \ C \downarrow f_{E_f}(\sigma) \sqcup \sigma'' \]

Recall that \( f_p \) is a transfer function and that \( f_{E_t} \) and \( f_{E_f} \) "filter" the store, e.g.,

\[ f_{x>2t}(x : 4, y : 3) = (x : 4, y : 3), \text{ whereas } f_{x>2t}(x : 0, y : 3) = \bot. \]

An example: if \( \text{Even}(x) \) \( (x=0) \) (while \( x \neq 3 \) \( (x = x+1) \))

\[ \langle x:1 \rangle \vdash \text{if } \text{Even}(x) \ (x=0) \ (\text{while } x \neq 3 \ (x = x+1)) \downarrow \bot \sqcup \langle x:3 \rangle = \langle x:3 \rangle \]

\[ \bot \vdash x = 0 \downarrow \bot \quad \langle x:1 \rangle \vdash \text{while } x \neq 3 \ldots \downarrow \bot \sqcup \langle x:3 \rangle \]

\[ \langle x:1 \rangle \vdash x = x + 1 \downarrow \langle x:2 \rangle \quad \langle x:2 \rangle \vdash \text{while } x \neq 3 \ldots \downarrow \bot \sqcup \langle x:3 \rangle \]

\[ \langle x:2 \rangle \vdash x = x + 1 \downarrow \langle x:3 \rangle \quad \langle x:3 \rangle \vdash \text{while } x \neq 3 \ldots \downarrow \langle x:3 \rangle \sqcup \bot = \langle x:3 \rangle \]

\[ \bot \vdash x = x + 1 \downarrow \bot \quad \bot \vdash \text{while } x \neq 3 \ldots \downarrow \bot \]
An abstract big-step derivation tree

Using the same inference rules but with abstract transfer functions for Parity = \{⊥, even, odd, ⊤\}, we generate an abstract tree that is infinite but regular:

\[
\begin{align*}
\langle x:odd \rangle \vdash & \text{ if Even}(x) \ (x = 0) \ (\text{while } x \neq 3 \ (x = x + 1)) \downarrow \bot \sqcup X \\
\bot & \vdash x = 0 \downarrow \bot \quad \langle x:odd \rangle \vdash \text{while } x \neq 3 \ldots \downarrow \langle x:odd \rangle \sqcup X = X \\
\langle x:odd \rangle & \vdash x = x + 1 \downarrow \langle x:even \rangle \quad \langle x:even \rangle \vdash \text{while } x \neq 3 \ldots \downarrow \bot \sqcup X \\
\langle x:even \rangle & \vdash x = x + 1 \downarrow \langle x:odd \rangle \quad \langle x:odd \rangle \vdash \text{while } x \neq 3 \ldots \downarrow X \\
& \ldots
\end{align*}
\]

Variable X denotes the answer from the repeated loop subderivation:

\[X = \langle x:odd \rangle \sqcup X\]

The least solution sets \(X = \langle x:odd \rangle\).
**Interprocedural analysis**

\[
\text{func } f(x) \text{ local } y; C. \quad [x \mapsto [E] \sigma][y \mapsto \bot] \vdash C \downarrow \sigma' \\
\sigma \vdash z = f(E) \downarrow \sigma[z \mapsto \sigma'(y)]
\]

where \([E] \sigma\) denotes \(E\)’s value with \(\sigma\), and \(x \mapsto v\) assigns \(v\) to \(x\).

**Example:**

\[
\text{func } g(x) \text{ local } z; z = x+1. \\
a = g(2); \ b = g(a); \ a = a*b
\]

\[
\langle a: \bot, b: \bot \rangle \vdash a = g(2); \ b = g(a); \ a = a*b \downarrow \langle a: \text{even}, b: \text{even} \rangle
\]

\[
\langle a: \bot, b: \bot \rangle \vdash a = g(2) \downarrow \langle a: \text{odd}, b: \bot \rangle \quad \langle a: \text{odd}, b: \bot \rangle \vdash b = g(a); \ a = a*b \downarrow \langle a: \text{even}, b: \text{even} \rangle
\]

\[
\langle x: \text{even}, z: \bot \rangle \vdash z = x + 1 \downarrow \langle x: \text{even}, z: \text{odd} \rangle \quad \langle a: \text{odd}, b: \text{even} \rangle \vdash a = a*b \downarrow \langle a: \text{even}, b: \text{even} \rangle
\]

\[
\langle a: \text{odd}, b: \bot \rangle \vdash b = g(a); \downarrow \langle a: \text{odd}, b: \text{even} \rangle
\]

\[
\langle x: \text{odd}, z: \bot \rangle \vdash z = x + 1 \downarrow \langle x: \text{odd}, z: \text{even} \rangle
\]

The derivation tree naturally separates the calling contexts.
"Too many" calling contexts (*) force widening (!):

\[
\text{func fac(a) local b; if } a = 0 \ (b = 1) \ (b = \text{fac}(a - 1); \ b = a \times b). \ c = \text{fac}(3)
\]

\[
\langle c : \bot \rangle \vdash c = \text{fac}(3) \downarrow \langle c : \top \rangle
\]

\[
\star \quad \langle 3, \bot \rangle \vdash \text{if } a = 0 \ (b = 1) \ (b = \text{fac}(a - 1); \ b = a \times b) \downarrow \perp \sqcup \langle T, T \rangle = \langle T, T \rangle
\]

\[
\bot \vdash b = 1 \downarrow \bot \quad \langle 3, \bot \rangle \vdash b = \text{fac}(a - 1); \ b = a \times b \downarrow \langle T, T \rangle
\]

\[
\langle 3, \bot \rangle \vdash b = \text{fac}(a - 1) \downarrow \langle 3, \top \rangle \quad 3, T \vdash b = a \times b \downarrow T, T
\]

\[
\star \quad \langle 3, \bot \rangle \sqcup \langle 2, \bot \rangle = \langle T, \bot \rangle \vdash \text{if } a = 0 \ldots \downarrow \langle 0, 1 \rangle \sqcup \langle T, T \ast X.b \rangle = X = \langle T, T \rangle
\]

\[
\langle 0, \bot \rangle \vdash b = 1 \downarrow \langle 0, 1 \rangle \quad \langle T, \bot \rangle \vdash b = \text{fac}(a - 1); \ b = a \times b \downarrow \langle T, T \ast X.b \rangle
\]

\[
\langle T, \bot \rangle \vdash b = \text{fac}(a - 1) \downarrow \langle T, X.b \rangle \quad \langle T, X.b \rangle \vdash b = a \times b \downarrow \langle T, T \ast X.b \rangle
\]

\[
\star! \quad \langle T, \bot \rangle \vdash \text{if } a = 0 \ldots \downarrow X
\]

\[
X = \langle 0, 1 \rangle \sqcup \langle T, T \ast X.b \rangle. \text{ The least solution sets } X = \langle T, T \rangle.
\]
Standard applications of static analysis
Abstract testing and model generation

\[ p_0 : \text{while isEven}(x) \{ \]
\[ p_1 : x = x \text{ div 2}; \]
\[ \} \]
\[ p_2 : x = 4 * x; \]
\[ p_3 : \text{exit} \]

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

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 \( \gamma(a_i) = \{s \mid B_i \text{ holds for } s\} \)

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*:

\[
\pi: \ldots ((\text{Rational})x).\text{ratValue}()\ldots
\]

A static analysis calculates the abstract store arriving at the cast at \(\pi\), a *checkpoint*:

- \(\pi, \langle \ldots x : \text{Int} \ldots \rangle\): no error possible — remove the run-time check (because \(\text{Int} \subseteq \text{Rational}\), hence \(\gamma(\text{Int}) \subseteq \gamma(\text{Rational})\)).

- \(\pi, \langle \ldots x : \text{Object} \ldots \rangle\): possible error — retain run-time check (because \(\text{Object} \nsubseteq \text{Rational}\))

- \(\pi, \langle \ldots x : \text{Bool} \ldots \rangle\): definite error, because \(\text{Bool} \cap \text{Rational} = \perp\) (assuming \(\gamma(\perp) = \emptyset\)).
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 \(a[e] \rightarrow e\) has value in range, \([0, a.length]\);
  for \(\text{int } x = e \rightarrow e\) has value in range, \([-2^{31} - 1, +2^{31} - 1]\)

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

♦ **Analysis:** constant propagation, where values are \(\{k\}, \bot, \text{ or } T\).

♦ **Checkpoints:**
  
  **uninitialized variables:** referenced variables have value \(\neq \bot\);
  **dead code:** at program point \(p_i\), arriving store has value \(\neq \bot\);
  **erroneous states:** at program point \(p_i : \text{Error}\), arriving store has value \(= \bot\). *(Note: This can be combined with a *backwards* analysis, starting from each \(p_i : \text{Error}\) with store \(T\), working backwards to see if an initial state is reached.)*
**Program transformation: Constant folding**

\[ p_0 : \ x = 1; \ y = 2; \]
\[ p_1 : \ \text{while} \ (x < y + z) \ {\}
    \[ p_2 : \ x = x + 1; \]
\}\
\[ p_3 : \ \text{exit} \]

The analysis tells us to replace \( y \) at \( p_1 \) by 2:
\[
\begin{align*}
x &= 1; \\
y &= 2; \\
\text{while} \ (x < 2 + z) \ x &= x + 1
\end{align*}
\]

**Basic principle of program transformation:**

*If \( a_i \in A \) arrives at point \( p_i : S \), where \( f_i : C \rightarrow C \) 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 \(-1, 0, 1, ...\) (but not \( \top \)).
Precondition checking and assertion synthesis

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

\[ p_0: \text{if } x=0 \]
\[ p_1: \text{then } x = x+1 \]
\[ p_2: \text{else } x = x-1 \]
\[ p_3: \text{halt } \langle x : \downarrow \text{notneg} \rangle \]

\[ x : \downarrow \top \cap \downarrow \text{notneg} = \downarrow \text{notneg} \]

where

\[ f^\#_{=0}(a) = a \cap \text{zero} = \alpha \circ f_{=0} \circ \gamma \]
\[ f^\#_{\neq 0} = \alpha \circ f_{\neq 0} \circ \gamma, \text{ e.g., } f^\#_{\neq 0}(\text{notneg}) = \text{pos}; \]
\[ f^\#_{\neq 0}(\text{zero}) = \bot; f^\#_{\neq 0}(\top) = \top \]
\[ f^\#_{+1} = \alpha \circ f_{+1} \circ \gamma, \text{ e.g., } f^\#_{+1}(\text{notneg}) = \text{pos} \]

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:

\[
\langle x : \text{notneg} \rangle \quad p_0 : \text{if } x=0 \\
\quad p_1 : x = x+1 \\
\quad p_2 : x = x-1 \\
\quad p_3 : \text{halt}
\]

The forwards-backwards analyses can be repeatedly alternated.

\[
f^\#_{\neq 0}(a) = a \cap \text{zero} = \alpha \circ f_{\neq 0} \circ \gamma \\
f^\#_{\neq 0} = \alpha \circ f_{\neq 0} \circ \gamma, \text{ e.g., } f^\#_{\neq 0}(\text{notneg}) = \text{pos}; \\
f^\#_{\neq 0}(\text{zero}) = \bot; f^\#_{\neq 0}(T) = T \\
f^\#_{+1} = \alpha \circ f_{+1} \circ \gamma, \text{ e.g., } f^\#_{+1}(\text{notneg}) = \text{pos}
\]
The “internal logic” of an abstract domain
Abstract values = logical propositions

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

$t S |= a$ iff $S \subseteq \gamma(a)$  e.g., $\{−3, −1\} |= neg$

t $a \vdash a'$ iff $a \sqsubseteq a'$  e.g., $neg \vdash any$

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

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

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

Read properties like $neg \in \text{Sign}$ as logical propositions, “isNegative”, etc.
A has an internal logic that \( \gamma \) 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 \( \gamma : A \rightarrow PC \) is an upper adjoint, then \( A \) has conjunction.

Proposition: When \( \gamma(\phi \sqcup \psi) = \gamma(\phi) \cup \gamma(\psi) \), then \( A \) has disjunction:

\[
S \models \phi \sqcup \psi \text{ iff } S \models \phi \text{ or } S \models \psi.
\]

Sign lacks disjunction: \( \gamma(\text{zero}) \not\models \text{neg} \sqcup \text{pos} \), because \( \text{neg} \sqcup \text{pos} = \text{any} \), but

\( \gamma(\text{zero}) \not\models \text{neg} \) and \( \gamma(\text{zero}) \not\models \text{pos} \).
Sometimes, we can implement a domain’s disjunctive completion \([\text{Cousots79,Giacobazzi00}]\):

\[
(P(\text{Int}), \subseteq) \langle \alpha_o, \gamma \rangle (P_\downarrow (\text{Sign}), \subseteq)
\]

\[
\gamma(T) = \bigcup_{a \in T} \gamma(a) \quad \alpha_o(S) = \downarrow \{ \alpha[c] \mid c \in S \}
\]

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

Now, \(\gamma\) preserves \(\cap\) and \(\cup\). Properties, \(a \in A\), are interpreted in \(P_\downarrow (A)\) as \(\alpha_o(\gamma(a)) = \downarrow \{ a \}\).

For \(A = P_\downarrow (\text{Sign})\), these assertions are exact:

\[
\phi ::= \text{neg} \mid \text{zero} \mid \text{pos} \mid \phi_1 \land \phi_2 \mid \phi_1 \lor \phi_2
\]
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

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

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

**Proposition:** If $A$ is a distributive complete lattice, $\sqcap$ is Scott-continuous, and upper adjoint $\gamma$ 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,

$$\phi ::= a \mid \phi_1 \cap \phi_2 \mid \phi_1 \lor \phi_2 \mid \phi_1 \Rightarrow \phi_2 \mid \bot$$

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

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

- $\gamma(a) = \text{given}$
- $\gamma(\phi \cap \psi) = \gamma(\phi) \cap \gamma(\psi)$
- $\gamma(\phi \lor \psi) = \gamma(\phi) \cup \gamma(\psi)$
- $\gamma(\phi \Rightarrow \psi) = \bigcup\{S \in \gamma[A] \mid S \cap \gamma(\phi) \subseteq \gamma(\psi)\}$
- $\gamma(\bot) = \emptyset$
**γ-completeness characterizes the internal logic**

The interpretation for conjunction, $\gamma(\phi \sqcap \psi) = \gamma(\phi) \cap \gamma(\psi)$, shows that $\gamma$-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 $f^#$ iff $f^#$ is $\gamma$-complete for $f^*$:

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

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

$$\phi ::= a \mid \phi_1 \sqcap \phi_2 \mid negate^#(\phi)$$

We can state “negate” assertions, e.g., $pos \models negate^#(neg \sqcap any)$. 
**Post-image (left-to-right) abstraction of relations**

\( f : C \rightarrow PC \) defines a relation in \( C \times C \), e.g., \( \{1, 3\}[\text{succ}][2, 4] \).

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

\[
post_f(S) = \bigcup_{c \in S} f(c).
\]

For Galois connection, \( PC(\overline{\alpha_o}, \overline{\gamma})P\downarrow(A) \), and \( f^\#: A \rightarrow P\downarrow(A) \),

\[\begin{align*}
\text{♦} & \text{ for } T \in P\downarrow(A), \text{ define } post_{f^#}(T) = \bigcup_{a \in T} f^#(a) = \bigcup_{a \in T} f^#(a) \\
\text{♦} & \text{ use } post_{f^#} \text{ to compute left-to-right (over)approximations of } f, \text{ e.g., } \{\text{neg}\}[\text{succ}^#]\{\text{neg, zero}\}, \text{ that is, neg } \lor \text{ zero.}
\end{align*}\]

**Proposition:** For \( f_{\text{best}}^# = \overline{\alpha_o} \circ f^* \circ \gamma \),

\[
(post_f)^#_{\text{best}} = \overline{\alpha_o} \circ post_f \circ \overline{\gamma} = post_{f_{\text{best}}^#}
\]

**Corollary:** If \( f \) is \( \gamma \)-complete, then \( (post_{f_{\text{best}}^#} \phi) \) is in \( P\downarrow(A) \)'s logic.
Given $\text{PC}(\alpha, \gamma)\downarrow A$, we have two relevant Galois connections between $\text{PC}$ and $\mathcal{P}_\downarrow(A)$.

Recall that $\overline{\gamma}(T) = \bigcup_{a \in T} \gamma(a)$ 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:**

$$\overline{\alpha_o}(S) = \bigcap\{T \mid S \subseteq \overline{\gamma}(T)\} = \downarrow\{\alpha\{c\} \mid c \in S\}$$

where

$$\downarrow T = \{\alpha \mid \text{exists } \alpha' \in T, \alpha \subseteq \alpha'\}.$$  

**Underapproximating abstraction:**

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

where

$$(D, \sqsubseteq_D)^{\text{op}}$$ 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., \( \{0, 1, 3\}[\text{succ}][1, 2, 4] \).

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

\[
\widetilde{\text{pre}}_f(S) = \bigcup \{S' \subseteq C \mid f^*(S') \subseteq S\} = \{c \mid f(c) \subseteq S\}
\]

For Galois connection, \( PC^{\text{op}}\langle \alpha_u, \nu \rangle \mathcal{P}_\uparrow(A)^{\text{op}} \) and \( f^\#: A \rightarrow \mathcal{P}_\downarrow(A) \),

\( \diamond \) for \( T \in \mathcal{P}_\downarrow(A) \), define \( \widetilde{\text{pre}}_{f^\#} = \{a \mid f^\#(a) \subseteq T\} \)

\( \diamond \) use \( \widetilde{\text{pre}}_{f^\#} \) to compute right-to-left (under)approximations of \( f \), e.g.,

\( \text{zero} \lor \text{pos}[\text{succ}^\#]\text{pos} \) and \( \text{none}[\text{succ}^\#]\text{zero} \) (!)

**Theorem:** \( (\widetilde{\text{pre}}_f)^\#_{\text{best}} = \overline{\alpha_u} \circ \widetilde{\text{pre}}_f \circ \overline{\nu} = \widetilde{\text{pre}}_{f^\#_{\text{best}}} \).

Because \( \widetilde{\text{pre}}_{f^\#_\phi} \) always underapproximates \( \widetilde{\text{pre}}_f(\overline{\nu}(\phi)) \), 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

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

Then, assert $a \vdash \phi$ iff $a \in [\phi]^A$.

This definition follows from the underapproximating Galois connection:

$$\bar{\gamma}(T) = \bigcup \{\gamma(a) \mid a \in T\}$$

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

That is, $[\phi]^A = \bar{\alpha}_u[\phi]$.

The inverted ordering gives underapproximation: $[\phi] \supseteq \overline{\gamma}( [\phi]^A )$. This form of external logic is standard in “abstract model checking.”
The inductively defined underapproximation to $\overline{\alpha_u}[\phi]$:

$$
\begin{align*}
&[[a]]_{\text{ind}}^A = \overline{\alpha_u}(\gamma(a)) \\
&[[\phi_1 \land \phi_2]]_{\text{ind}}^A = [[\phi_1]]_{\text{ind}}^A \cap [[\phi_2]]_{\text{ind}}^A \\
&[[\phi_1 \lor \phi_2]]_{\text{ind}}^A = [[\phi_1]]_{\text{ind}}^A \cup [[\phi_2]]_{\text{ind}}^A \\
&[[[f] \phi]]_{\text{ind}}^A = \overline{\text{pre}}_{f^\#}[[\phi]]_{\text{ind}}^A = \{ a \in A \mid f^\#(a) \in [[\phi]]_{\text{ind}}^A \}
\end{align*}
$$

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

Soundness ($\vdash$ implies $\models$) is immediate, and completeness ($\models$ implies $\vdash$) follows when $\overline{\alpha_u} \circ [\cdot] = [[\cdot]]_{\text{ind}}^A$. 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

\( B = \{ \bot, \top \} \), where \( \top \) means "might normalize" and \( \bot \) means "does not normalize".

\[
B \rightarrow B
\]

\[
[3] = \lambda x. \top
\]

\[
[2] = \lambda x. x
\]

\[
[1] = \lambda x. \bot
\]

\[
B \rightarrow B \rightarrow B
\]

\[
\text{where } [m] \cdot \top = [m] \quad \text{and } [m] \cdot \bot = [n]
\]

Example: \( F\ m\ n = \text{if } (m=0) (n) (F\ (m+1)\ n) \)

\( F^\# = \lambda a: B. \lambda b: B. a \sqcap (b \sqcup (F^\# \ a \ b)) \)

\( \text{graph}(F^\#) = \{ \bot \mapsto \bot \mapsto \bot, \bot \mapsto \top \mapsto \bot, \top \mapsto \bot \mapsto \bot, \top \mapsto \top \mapsto \top \}\).

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:** \( F^\# = \lambda f : B \to (B \to B).\lambda x : B. (x, f \cdot x) \)

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

\[
\text{graph}(F^\#) = \{
\begin{array}{c}
\{ \\
[1] \mapsto \bot \mapsto (\bot, \bot), \\
[1] \mapsto \top \mapsto (\top, \bot), \\
[2] \mapsto \bot \mapsto (\bot, \bot), \\
[2] \mapsto \top \mapsto (\top, \top), \\
\ldots \\
[3] \mapsto \bot \mapsto (\bot, \top), \\
[3] \mapsto \top \mapsto (\top, \top)
\end{array}
\}
\]

It’s model-checking-like and feasible to implement!
Assemble the graph in increments and retain only useful (“frontier”) entries, as based on these consequences of monotonicity:

♦ if \( a \mapsto b \in \text{frontier}(F^\#) \), then (i) for all \( a' \sqsubseteq a \), \( a' \mapsto b \) is sound; (ii) for all \( b' \sqsubseteq b \), \( a \mapsto b' \) is sound.

♦ if \( a \mapsto \top \in \text{frontier}(F^\#) \), then for all \( a' \sqsupseteq a \), \( a' \mapsto \top \) is sound.

♦ if \( a_1 \mapsto b_1, a_2 \mapsto b_2 \in \text{frontier}(F^\#) \), then (i) \( a_1 \sqcap a_2 \mapsto b_1 \sqcap b_2 \) is sound; (ii) if \( F^\# \) preserves \( \sqcup \) (holds when \( F^\# \)’s domain is a disjunctive completion), then \( a_1 \sqcup a_2 \mapsto b_1 \sqcup b_2 \) is sound.

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

\[
\text{frontier}(F^\#) = \{ [\frac{2}{2}] \mapsto \bot \mapsto (\bot, [2]), [\frac{2}{2}] \mapsto \top \mapsto (\top, [2]), [\frac{3}{1}] \mapsto \top \mapsto (\top, [3]) \} 
\]
Example inferences based on the frontier

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

\[
\text{frontier}(F^#) = \{
\begin{align*}
[2] & \mapsto \bot \mapsto (\bot, [2]), \\
[2] & \mapsto \top \mapsto (\top, [2]), \\
[3] & \mapsto \top \mapsto (\top, [3])
\end{align*}
\]

we can conclude

\[
\begin{align*}
[2] & \mapsto \top \mapsto (\top, [3]) \quad \text{is sound} \quad (\text{because } [2] \subseteq [3]) \\
[3] & \mapsto \top \mapsto (\top, [3]) \quad \text{is sound} \quad (\text{because } [3], \top \text{ map to } (\top, [3])) \\
[2] & \mapsto \top \mapsto (\top, [2]) \quad \text{is sound} \quad (\text{because } [2] = [2] \cap [3]) \\
[3] & \mapsto \top \mapsto (\top, [3]) \quad \text{is sound} \quad (\text{because } [3] = [2] \cup [3])
\end{align*}
\]
Integrating symbolic evaluation with frontiers

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

\[
\text{symbolicFrontier}(F^\#) = \{ \\
[2] \mapsto a \mapsto (a, a), \\
[3] \mapsto a \mapsto (a, [3] \cdot a), \\
f \mapsto a \mapsto (a, f \cdot a) \} \\
\]

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: \[ F \ x = \mathrm{if} \ (\ldots) \ (g \ x) \ (h(F(f \ x))) \]

\[ \mathcal{F}^\# = \bigsqcup_{i \geq 0} F_i, \quad \text{where} \quad F_0 = \lambda a. \bot \]
\[ F_{i+1} = \lambda a. (g \ a) \sqcup (h(F_i(f \ a))) \]

By inductive reasoning,

\[ F_i = \bigsqcup_{0 \leq j < i} h^j(g(f^j a)) \]
\[ \subseteq \bigsqcup_{0 \leq j < i} h^j(g(f^* a)) \]
\[ \subseteq h^*(g(f^* a)) \]

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


