Abstract Interpretation

Lecture 5
History

• **One breakthrough paper**
  - Cousot & Cousot ‘77 (?)

• **Inspired by**
  - Dataflow analysis
  - Denotational semantics

• **Enthusiastically embraced by the community**
  - At least the functional community . . .
  - At least the first half of the paper . . .
A Tiny Language

- Consider a language with only integers and multiplication.

\[ e = i \mid e \ast e \]

\[ \mu: \text{Exp} \rightarrow \text{Int} \]

\[ \mu(i) = i \]

\[ \mu(e_1 \ast e_2) = \mu(e_1) \times \mu(e_2) \]
An Abstraction

- Define an abstract semantics that computes only the sign of the result.

\[ \sigma: \text{Exp} \rightarrow \{+,-,0\} \]

\[
\begin{align*}
\sigma(i) &= \begin{cases} 
+ & \text{if } i > 0 \\
0 & \text{if } i = 0 \\
- & \text{if } i < 0
\end{cases} \\
\sigma(e_1 \times e_2) &= \sigma(e_1) \times \sigma(e_2)
\end{align*}
\]

\[
\begin{array}{c|ccc}
\times & + & 0 & - \\
\hline
+ & + & 0 & - \\
0 & 0 & 0 & 0 \\
- & - & 0 & +
\end{array}
\]
Soundness

• We can show that this abstraction is correct in the sense that it correctly predicts the sign of an expression.
• Proof is by structural induction on $e$.

\[
\begin{align*}
\mu(e) > 0 & \iff \sigma(e) = + \\
\mu(e) = 0 & \iff \sigma(e) = 0 \\
\mu(e) < 0 & \iff \sigma(e) = -
\end{align*}
\]
Another View of Soundness

- The soundness proof is clunky
  - each case repeats the same idea.
- Instead, directly associate each abstract value with the set of concrete values it represents.

\[ \gamma : \{+, 0, -\} \rightarrow 2^{\text{Int}} \]

\[ \gamma(+) = \{i \mid i > 0\} \]
\[ \gamma(0) = \{0\} \]
\[ \gamma(-) = \{i \mid i < 0\} \]
Another View (Cont.)

- The concretization function
  - Mapping from abstract values to (sets of) concrete values
- Let
  - \( D \) be the concrete domain,
  - \( A \) the abstract domain.

\[
\mu(e) \in \gamma(\sigma(e))
\]
Abstract Interpretation

• This is an *abstract interpretation*.
  - Computation in an *abstract domain*
  - In this case \{+,-,0\}.

• The abstract semantics is sound
  - approximates the standard semantics.

• The concretization function establishes the connection between the two domains.
Adding

- Extend our language with unary -

\[ \mu(-e) = -\mu(e) \]
\[ \sigma(-e) = -\sigma(e) \]
Adding $+$

- Adding addition is not so easy.
- The abstract values are not closed under addition.

\[
\begin{align*}
\mu(e_1 + e_2) &= \mu(e_1) + \mu(e_2) \\
\sigma(e_1 + e_2) &= \sigma(e_1) + \sigma(e_2)
\end{align*}
\]
Solution

• We need another abstract value to represent a result that can be any integer.
• Finding a domain closed under all the abstract operations is often a key design problem.

\[ \gamma(T) = \text{Int} \]

<table>
<thead>
<tr>
<th></th>
<th>+</th>
<th>O</th>
<th>-</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>+</td>
<td>T</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>O</td>
<td>+</td>
<td>O</td>
<td>-</td>
<td>T</td>
</tr>
<tr>
<td>-</td>
<td>T</td>
<td>-</td>
<td>-</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>
Extending Other Operations

- We also need to extend the other abstract operations to work with $T$.

\[
\begin{array}{c|cccc}
\times & + & 0 & - & T \\
\hline
+ & + & 0 & - & T \\
0 & 0 & 0 & 0 & 0 \\
- & - & 0 & + & T \\
T & T & T & 0 & T \\
\end{array}
\]
Examples

Abstract computation loses information

\[ \mu((1 + 2) + -3) = 0 \]
\[ \sigma((1 + 2) + -3) = (+ + +) + (-+ +) = T \]

No loss of information

\[ \mu((5 * 5) + 6) = 31 \]
\[ \sigma((5 * 5) + 6) = (+ \times +) + + = + \]
Adding / (Integer Division)

• Adding / is straightforward except for the case of division by 0.
• If we divide each integer in a set by 0, what set of integers results? The empty set.

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

<table>
<thead>
<tr>
<th></th>
<th>+</th>
<th>0</th>
<th>-</th>
<th>T</th>
<th>\bot</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td>\bot</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>\bot</td>
</tr>
<tr>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>\bot</td>
</tr>
<tr>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>\bot</td>
</tr>
<tr>
<td>\bot</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>\bot</td>
</tr>
</tbody>
</table>

Profs. Aiken, Barrett & Dill  CS 357  Lecture 5
Adding / (Cont.)

• As before we need to extend the other abstract operations.
• In this case, every entry involving bottom is bottom
  - all operations are *strict* in bottom

\[
\begin{align*}
\bot + x &= \bot \\
\bot \times \bot &= \bot \\
- \bot &= \bot
\end{align*}
\]
The Abstract Domain

- Our abstract domain forms a lattice.
  - A partial order \( x \leq y \Leftrightarrow \gamma(x) \subseteq \gamma(y) \)
  - Every finite subset has a least upper bound (lub) & greatest lower bound (glb).

- We write \( A \) for an abstract domain
  - a set of values + an ordering
Lattice Lingo

• A lattice is *complete* if every subset (finite or infinite) has lub’s and glb’s.
  - Every finite lattice is complete

• Thus every lattice has a top/bottom element.
  - Usually needed in abstract interpretations.
The Abstraction Function

• The abstraction function maps concrete values to abstract values.
  - The dual of concretization.
  - The smallest value of \( A \) that is the abstraction of a set of concrete values.

\[
\alpha : 2^{\text{Int}} \rightarrow A \\
\alpha(S) = \text{lub}\left(\{- | i < 0 \land i \in S\}, \{0 | 0 \in S\}, \{+ | i > 0 \land i \in S\}\right)
\]
A General Definition

- An abstract interpretation consists of
  - An abstract domain $A$ and concrete domain $D$
  - Concretization and abstraction functions forming a Galois insertion.
  - A (sound) abstract semantic function.

Galois insertion:

$$\forall x \in 2^D. \ x \subseteq \gamma(\alpha(x))$$

$$\forall a \in A. \ x = \alpha(\gamma(x))$$

or

$$id \leq \gamma \circ \alpha$$

$$id = \alpha \circ \gamma$$
Galois Insertions

- The abstract domain can be thought of as dividing the concrete domain into subsets (not disjoint).
- The abstraction function maps a subset of the domain to the smallest containing abstract value.

\[
\begin{align*}
\text{id} & \leq \gamma \circ \alpha \\
\text{id} & = \alpha \circ \gamma
\end{align*}
\]
In correct abstract interpretations, we expect the following diagram to commute.
General Conditions for Correctness

Three conditions guarantee correctness in general:

\( \alpha \) and \( \gamma \) form a Galois insertion
\[
\text{id} \leq \gamma \circ \alpha , \quad \text{id} = \alpha \circ \gamma
\]
\( \alpha \) and \( \gamma \) are monotonic
\[
\alpha(\alpha(x)) \leq \alpha(\alpha(y))
\]
Abstract operations \( \text{op} \) are locally correct:
\[
\gamma(\text{op}(s_1,\ldots,s_n)) \supseteq \text{op}(\gamma(s_1),\ldots,\gamma(s_n))
\]
Generic Correctness Proof

Proof by induction on the structure of $e$: $\mu(e) \in \gamma(\sigma(e))$

\[
\begin{align*}
\mu(e_1 \text{ op } e_2) \\
= & \quad \mu(e_1) \text{ op } \mu(e_2) \quad \text{def. of } \mu \\
\in & \quad \gamma(\sigma(e_1)) \text{ op } \gamma(\sigma(e_2)) \quad \text{by induction} \\
\subseteq & \quad \gamma(\sigma(e_1) \text{ op } \sigma(e_2)) \quad \text{local correctness} \\
= & \quad \gamma(\sigma(e_1 \text{ op } e_2)) \quad \text{def of } \sigma
\end{align*}
\]
A Second Notion of Correctness

- We can define correctness using abstraction instead of concretization.

\[ \mu(e) \in \gamma(\sigma(e)) \equiv \alpha(\{\mu(e)\}) \leq \sigma(e) \]

\[ \Rightarrow \text{ direction} \]

\[ \mu(e) \in \gamma(\sigma(e)) \]

\[ \alpha(\{\mu(e)\}) \leq \alpha(\gamma(\sigma(e))) \quad \text{monotonicity} \]

\[ \alpha(\{\mu(e)\}) \leq \sigma(e) \quad \alpha \circ \gamma = id \]
Correctness (Cont.)

• The other direction . . .

\[
\mu(e) \in \gamma(\sigma(e)) \equiv \alpha(\{\mu(e)\}) \leq \sigma(e)
\]

\(\Leftarrow\) direction

\[
\alpha(\{\mu(e)\}) \leq \sigma(e)
\]

\[
\gamma(\alpha(\{\mu(e)\})) \leq \gamma(\sigma(e)) \quad \text{monotonicity}
\]

\[
\mu(e) \in \gamma(\sigma(e)) \quad \text{id} \leq \gamma \circ \alpha
\]
A Language with Input

• The next step is to add language features besides new operations.
• We begin with input, modeled as a single free variable $x$ in expressions.

$$e = i \mid e \ast e \mid -e \mid \ldots \mid x$$
Semantics

• The meaning function now has type

\[ \mu : \text{Exp} \rightarrow \text{Int} \rightarrow \text{Int} \]

• We write the function curried with the expression as a subscript.

\[
\begin{align*}
\mu_i(j) &= i \\
\mu_x(j) &= j \\
\mu_{e_1 \cdot e_2}(j) &= \mu_{e_1}(j) \cdot \mu_{e_2}(j) \\
\mu_{e_1 + e_2}(j) &= \mu_{e_1}(j) + \mu_{e_2}(j) \\
\ldots &= \ldots
\end{align*}
\]
Abstract Semantics

- **Abstract semantic function:**
  \[ \sigma : \text{Exp} \rightarrow A \rightarrow A \]

- Also write this semantics curried.
  \[
  \begin{align*}
  \sigma_i(\bar{j}) &= \bar{i} \\
  \sigma_x(\bar{j}) &= \bar{j} \\
  \sigma_{e_1 * e_2}(\bar{j}) &= \sigma_{e_1}(\bar{j}) * \sigma_{e_2}(\bar{j}) \\
  \sigma_{e_1 + e_2}(\bar{j}) &= \sigma_{e_1}(\bar{j}) + \sigma_{e_2}(\bar{j}) \\
  \vdots &= \vdots \\
  \bar{i} &= \alpha(\{i\})
  \end{align*}
  \]
Correctness

• The correctness condition needs to be generalized.
• This is the first real use of the abstraction function.
• The following are all equivalent:

∀ i. \( \mu_e(i) \in \gamma(\sigma_e(\alpha(\{i\}))) \)

\( \mu_e \leq_D \gamma \circ \sigma_e \circ \alpha \)

\( \alpha \circ \mu_e \leq_A \sigma_e \circ \alpha \)
Local Correctness

• We also need a modified local correctness condition.

\[ \text{op}(\gamma(\sigma_{e_1}(\bar{j})), \ldots, \gamma(\sigma_{e_n}(\bar{j}))) \subseteq \gamma(\text{op}(\sigma_{e_1}(\bar{j}), \ldots, \sigma_{e_n}(\bar{j}))) \]
Proof of Correctness

Thm $\mu_e(j) \in \gamma(\sigma_e(\bar{j}))$

Proof (by induction)

Basis. $\mu_i(j) = i \in \gamma(\bar{i}) = \gamma(\sigma_i(\bar{j}))$
$\mu_x(j) = j \in \gamma(\bar{j}) = \gamma(\sigma_x(\bar{j}))$

Step

$\mu_{op(e_1,\ldots,e_n)}(j)$
$= \text{op}(\mu_{e_1}(j),\ldots,\mu_{e_n}(j))$ \hspace{1cm} \text{def. of } \mu$
$\subseteq \text{op}(\gamma(\sigma_{e_1}(\bar{j})),\ldots,\gamma(\sigma_{e_n}(\bar{j})))$ \hspace{1cm} \text{induction}$
$\subseteq \gamma(\text{op}(\sigma_{e_1}(\bar{j}),\ldots,\sigma_{e_n}(\bar{j})))$ \hspace{1cm} \text{local correctness}$
$= \gamma(\sigma_{op(e_1,\ldots,e_n)}(\bar{j}))$ \hspace{1cm} \text{def. of } \sigma$
If-Then-Else

\[ e = \ldots | \text{if } e = e \text{ then } e \text{ else } e \mid \ldots \]

\[ \mu_{\text{if } e_1 = e_2 \text{ then } e_3 \text{ else } e_4} (i) = \begin{cases} 
\mu_{e_3} (i) & \text{if } \mu_{e_1} (i) = \mu_{e_2} (i) \\
\mu_{e_4} (i) & \text{if } \mu_{e_1} (i) \neq \mu_{e_2} (i) 
\end{cases} \]

\[ \sigma_{\text{if } e_1 = e_2 \text{ then } e_3 \text{ else } e_4} (\bar{i}) = \sigma_{e_3} (\bar{i}) \sqcup \sigma_{e_4} (\bar{i}) \]

• Note the lub operation in the abstract function; this is why we need lattices as domains.
Correctness of If-Then-Else

• Assume the true branch is taken.
• (The argument for the false branch is symmetric.)

\[
\mu_{e_3}(i) \\
\subseteq \gamma(\sigma_{e_3}(\bar{i})) \quad \text{by induction} \\
\subseteq \gamma(\sigma_{e_3}(\bar{i})) \cup \gamma(\sigma_{e_4}(\bar{i})) \\
\subseteq \gamma(\sigma_{e_3}(\bar{i}) \cup \sigma_{e_4}(\bar{i})) \quad \text{monotonicity of } \gamma
\]
Recursion

- Add recursive definitions
  - of a single variable for simplicity
- The semantic function is

\[ \mu : \text{Exp} \rightarrow \text{Int} \rightarrow \text{Int}_\perp \]

\[ \text{program} = \text{def } f(x) = e \]
\[ e = \ldots | f(e) \]
Revised Meaning Function

- Define an auxiliary semantics taking a function (for the free variable $\mathcal{A}$) and an integer (for $x$).

$$
\mu': \text{Exp} \rightarrow (\text{Int} \rightarrow \text{Int}_\perp) \rightarrow \text{Int} \rightarrow \text{Int}_\perp
$$

$$
\mu'_{f(e)}(g)(j) = g(\mu'_{e}(g)(j))
$$

$$
\mu'_{x}(g)(j) = j
$$

$$
\mu'_{e_1+e_2}(g)(j) = \mu'_{e_1}(g)(j) + \mu'_{e_2}(g)(j)
$$
Meaning of Recursive Functions

\[ \mu : \text{Exp} \rightarrow \text{Int} \rightarrow \text{Int}_\bot \]
\[ \mu' : \text{Exp} \rightarrow (\text{Int} \rightarrow \text{Int}_\bot) \rightarrow \text{Int} \rightarrow \text{Int}_\bot \]

Consider a function \( \text{def } f = e \)

Define an ascending chain \( f_0, f_1, \ldots \) in \( \text{Int} \rightarrow \text{Int}_\bot \)

\[ f_0 = \lambda x. \bot \]
\[ f_{i+1} = \mu'_e(f_i) \]

Define \( \mu_f = \bigcup_{i} f_i \)
Abstract Semantics Revised

- Define an analogous auxiliary function for the abstract semantics.

\[ \sigma': \text{Exp} \rightarrow (A \rightarrow A) \rightarrow A \rightarrow A \]

\[ \sigma'_{f(e)}(g)(i) = g(\sigma'_{e}(g)(i)) \]

\[ \sigma'_{\times}(g)(i) = i \]

\[ \sigma'_{e_1+e_2}(g)(i) = \sigma'_{e_1}(g)(i) + \sigma'_{e_2}(g)(i) \]
Abstract Semantics Revised II

• We need one more condition for the abstract semantics.

• All abstract functions are required to be monotonic.

• Thm. Any monotonic function on a complete lattice has a least fixed point.
Abstract Meaning of Recursion

\[ \sigma : \text{Exp} \rightarrow A \rightarrow A \]
\[ \sigma' : \text{Exp} \rightarrow (A \rightarrow A) \rightarrow A \rightarrow A \]

Consider a function \( \text{def } f = e \)

Define an ascending chain \( \overline{f}_0, \overline{f}_1, \ldots \) in \( A \rightarrow A \)
\[ \overline{f}_0 = \lambda a. \perp \]
\[ \overline{f}_{i+1} = \sigma'_e(\overline{f}_i) \]

Define \( \sigma_f = \bigcup_i \overline{f}_i \)
Correctness

Corresponding elements of the chain stand in the correct relationship.
Correctness (Cont.)

\[ \forall i. \ f_i(j) \in \gamma(f_i(j)) \]
\[ \Rightarrow \bigcup_{i \geq 0} f_i(j) \in \bigcup_{i \geq 0} \gamma(f_i(j)) \quad \text{chains stabilize} \]
\[ \Rightarrow \bigcup_{i \geq 0} f_i(j) \in \gamma\left(\bigcup_{i \geq 0} f_i(j)\right) \quad \text{monotonicity of } \gamma \]
\[ \Rightarrow \mu_f(j) \in \gamma(\sigma_f(j)) \quad \text{by definition} \]
Example

\[
\text{def } f(x) = \begin{cases} 
1 & \text{if } x = 0 \\
 x \cdot f(x + 1) & \text{else}
\end{cases}
\]

**Abstraction:**

\[
\text{lfp}\left(\sigma'(\text{if } x = 0 \text{ then } 1 \text{ else } x \cdot f(x + 1))\right)
\]

**Simplified:**

\[
\text{lfp}\left(\lambda f. \lambda x. + \cup (\bar{x} \cdot \bar{x} \cdot \bar{f}(\bar{x} + -))\right)
\]
Strictness

• We will assume our language is strict.
  – Makes little difference in quality of analysis for this example.
• Assume that $f(\bot) = \bot$
• Therefore it is sound to define $\overline{f}(\bot) = \bot$
Calculating the LFP

\[
\text{lfp}\left(\lambda f.\lambda x. x \lor \bar{x} \lor f(\bar{x} + -)\right)
\]

\[
\bar{f}_0 = \begin{array}{cccc}
\bot & - & 0 & + \\
\bot & \bot & \bot & \bot
\end{array}
\]

\[
\bar{f}_1 = \begin{array}{cccc}
\bot & - & 0 & + \\
\bot & + & + & +
\end{array}
\]

\[
\bar{f}_2 = \begin{array}{cccc}
\bot & - & 0 & + \\
\bot & T & T & +
\end{array}
\]

\[
\bar{f}_3 = \begin{array}{cccc}
\bot & - & 0 & + \\
\bot & T & T & T
\end{array}
\]
Notes

• In this case, the abstraction yields no useful information!

• Note that sequence of functions forms a strictly ascending chain until stabilization

\[ f_0 < f_1 < f_2 < f_3 = f_4 = f_5 = ... \]

• But the sequence of values at particular points may not be strictly ascending:

\[ f_0(+) < f_1(+) = f_2(+) < f_3(+) = f_4(+) = f_5(+) = ... \]
Notes (Cont.)

- Lesson: The fixed point is being computed in the domain $(A \rightarrow A) \rightarrow A \rightarrow A$

- The fixed point is not being computed in $A \rightarrow A$

- Make sure you check the domain of the fixed point operator.
Strictness Analysis
Strictness Analysis Overview

• In lazy functional languages, it may be desirable to change *call-by-need* (lazy evaluation) to *call-by-value*.

• CBN requires building “thunks” (closures) to capture the lexical environment of unevaluated expressions.

• CBV evaluates its argument immediately, which is wasteful (or even wrong) if the argument is never evaluated under CBN.
Correctness

- Substituting CBV for CBN is always correct if we somehow know that a function evaluates its argument(s).

- A function \( f \) is strict if \( f(\perp) = \perp \)

- Observation: if \( f \) is strict, then it is correct to pass arguments to \( f \) by value.
Outline

• Deciding whether a function is strict is undecidable.

• Mycroft’s idea: Use abstract interpretation.

• Correctness condition: If \( f \) is non-strict, we must report that it is non-strict.
The Abstract Domain

- Continue working with the same language (1 recursive function of 1 variable).
- New abstract domain 2:
Concretization/Abstraction

- The concretization/abstraction functions say
  - 0 means the computation definitely diverges
  - 1 means nothing is known about the computation
  - $D$ is the concrete domain

\[
\begin{align*}
\gamma(0) &= \{\bot\} \quad \alpha(\{\bot\}) = 0 \\
\gamma(1) &= D \quad \alpha(S) = 1 \text{ if } S \neq \{\bot\}
\end{align*}
\]
Abstract Semantics

• Next step is to define an abstract semantics

• Transform $f: \text{Int} \rightarrow \text{Int}$ to $\bar{f}: 2 \rightarrow 2$

• Transform values $v: \text{Int}$ to $\bar{v}: 2$

• To test strictness check if $\bar{f}(0) = 0$
Abstract Semantics (Cont.)

• An $a$ stands for an abstract value (0 or 1).
• Treat 0,1 as false, true respectively.

\[ \sigma'_{x}(g)(a) = a \]
\[ \sigma'_{i}(g)(a) = 1 \]
\[ \sigma'_{e}(g)(a) = \sigma'_{e}(g)(a) \]
\[ \sigma'_{e_{1}e_{2}}(g)(a) = \sigma'_{e_{1}}(g)(a) \land \sigma'_{e_{2}}(g)(a) \]
\[ \sigma'_{f(e)}(g)(a) = g(\sigma'_{e}(g)(a)) \]
The Rest of the Rules

\[ \sigma'_{e_1+e_2}(g)(a) = \sigma'_{e_1}(g)(a) \land \sigma'_{e_2}(g)(a) \]

\[ \sigma'_{e_1/e_2}(g)(a) = \sigma'_{e_1}(g)(a) \land \sigma'_{e_2}(g)(a) \]

\[ \sigma'_{\text{if } e_1 = e_2 \text{ then } e_3 \text{ else } e_4}(g)(a) = \sigma'_{e_1}(g)(a) \land \sigma'_{e_2}(g)(a) \land (\sigma'_{e_3}(g)(a) \lor \sigma'_{e_4}(g)(a)) \]

\[ \sigma_{\text{def } f = e} = \text{Ifp } \sigma'_{e} \]
An Example

def f(x) = if x = 0 then 1 else x * f(x-1)

lfp(\sigma'(if x = 0 then 1 else x * f(x+1)))

lfp(\lambda f.\lambda x.x) = \lambda a.a

(\lambda a.a) 0 = 0       The function is strict in x.
Calculating the LFP

\[ \text{lfp}\left( \lambda \bar{f}. \lambda \bar{x}. \bar{x} \wedge 1 \wedge (1 \lor (\bar{x} \land \bar{f}(\bar{x} \land 1))) \right) \]

\[
\begin{array}{c|c}
\bar{f}_0 & 0 & 1 \\
0 & 0 & 0 \\
\end{array}
\]

\[
\begin{array}{c|c}
\bar{f}_1 & 0 & 1 \\
0 & 0 & 1 \\
\end{array}
\]

\[
\begin{array}{c|c}
\bar{f}_2 & 0 & 1 \\
0 & 0 & 1 \\
\end{array}
\]
Another Example

• Generalize to recursive functions of two variables.

\[
def f(x,y) = \begin{cases} 
0 & \text{if } x = 0 \\
\text{if } x > 0 & \text{then } f(x-1, f(x,y)) 
\end{cases}
\]

\[\text{lfp}(\sigma'(\text{if } x = 0 \text{ then } 0 \text{ else } f(x+1, f(x,y)))) = \]

\[\text{lfp}(\lambda f. \lambda (\bar{x}, \bar{y}). \bar{x} \land 1 \land (1 \lor \ldots)) = \]

\[\lambda (\bar{x}, \bar{y}). \bar{x}\]
Example (Cont.)

- For multi-argument functions, check each argument combination of the form \((1,\ldots,1,0,1,\ldots,1)\).

\[
\begin{align*}
(\lambda(x,y). \overline{x}) (0,1) &= 0 & \text{\(x\) can be passed by value.} \\
(\lambda(x,y). \overline{x}) (1,0) &= 1 & \text{Unsafe to pass \(y\) by value.}
\end{align*}
\]
Summary of Strictness Analysis

• Mycroft’s technique is sound and practical.
  - Widely implemented for lazy functional languages.
  - Makes modest improvement in performance (a few %).
  - The theory of abstract interpretation is critical here.

• Mycroft’s technique treats all values as atomic.
  - No refinement for components of lists, tuples, etc.

• Many research papers take up improvements for data types, higher-order functions, etc.
  - Most of these are very slow.
Conclusions

- The Cousot&Cousot paper(s) generated an enormous amount of other research.
- Abstract interpretation as a theory and abstract interpretation as a method of constructing tools are often confused.
- Slogan of most researchers:

Finite Lattices + Monotonic Functions = Program Analysis
Where is Abstract Interpretation Weak?

- Theory is completely general

- The part of the original paper people understand is limited
  - Finite domains + monotonic functions
Data Structures and the Heap

• Requires a finite abstraction
  - Which may be tuned to the program
  - More often is “empty list, list of length 1, unknown length”

• Similar comments apply to analyzing heap properties
  - E.g., a cell has 0 references, 1 references, many references
Size of Domains

- Large domains = slow analysis

- In practice, domains are forced to be small
  - Chain height is the critical measure

- The focus in abstract interpretation is on correctness
  - Not much insight into efficient algorithms
Context Sensitivity

• No particular insight into context sensitivity

• Any reasonable technique is an abstract interpretation
Higher-Order Functions

• **Makes clear how to handle higher-order functions**
  - Model as abstract, finite functions
  - Ordering on functions is pointwise
    • Problem: huge domains

• **Break with the dependence on control-flow graphs**
Forwards vs. Backwards

• The forwards vs. backwards mentality permeates much of the abstract interpretation literature

• But nothing in the theory says it has to be that way