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.

$$e = i \mid e * e$$

$$\mu : Exp \to Int$$

$$\mu(i) = i$$

$$\mu(e_1 * e_2) = \mu(e_1) \times \mu(e_2)$$

An Abstraction

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

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

$$\sigma(i) = \begin{cases} + & \text{if } i > 0 \\ 0 & \text{if } i = 0 \\ - & \text{if } i < 0 \end{cases}$$

$$\sigma(e_1 * e_2) = \sigma(e_1) \mathbin{\bar{\times}} \sigma(e_2)$$

$\bar{\times}$	$+$	0	$-$
$+$	$+$	0	$-$
0	0	0	0
$-$	$-$	0	$+$

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.

$$\mu(e) > 0 \Leftrightarrow \sigma(e) = +$$

$$\mu(e) = 0 \Leftrightarrow \sigma(e) = 0$$

$$\mu(e) < 0 \Leftrightarrow \sigma(e) = -$$

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.

$$\gamma : \{+, 0, -\} \to 2^{Int}$$

$$\begin{aligned}
\gamma(+) &= \{i \mid i > 0\} \\
\gamma(0) &= \{0\} \\
\gamma(-) &= \{i \mid i < 0\}
\end{aligned}$$

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.

$$\begin{tikzcd} & & A \arrow[d, "\gamma"] \\ \mathrm{Exp} \arrow[urr, "\sigma"] \arrow[rr] & & \mu \in 2^D \end{tikzcd}$$

$$\mu(e) \in \gamma(\sigma(e))$$

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 -

$$\mu(-e) = -\mu(e)$$

$$\sigma(-e) = \overline{-}\,\sigma(e)$$

$\overline{-}$	$+$	0	$-$
	$-$	0	$+$

Adding +

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

$$\mu(e_1 + e_2) = \mu(e_1) + \mu(e_2)$$

$$\sigma(e_1 + e_2) = \sigma(e_1) \,\bar{+}\, \sigma(e_2)$$

$\bar{+}$	+	0	–
+	+	+	?
0	+	0	–
–	?	–	–

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.

$\gamma(\top) = \text{Int}$

$\overline{+}$	$+$	0	$-$	$\top$
$+$	$+$	$+$	$\top$	$\top$
0	$+$	0	$-$	$\top$
$-$	$\top$	$-$	$-$	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

Extending Other Operations

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

$\bar{\times}$	+	0	−	T
+	+	0	−	T
0	0	0	0	0
−	−	0	+	T
T	T	0	T	T

$\bar{-}$	+	0	−	T
	−	0	+	T

Examples

Abstract computation loses information

$$\mu((1+2)+-3) \quad = \quad 0$$

$$\sigma((1+2)+-3) \quad = \quad (+ \;\; \bar{+} \;\; +) \;\; \bar{+} \;\; (\bar{-}+) = \mathsf{T}$$

No loss of information

$$\mu((5*5)+6) \quad = \quad 31$$

$$\sigma((5*5)+6) \quad = \quad (+ \;\; \bar{\times} \;\; +) \;\; \bar{+} \;\; + = +$$

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.

$$\gamma(\perp) = \emptyset$$

$\overline{/}$	$+$	0	$-$	$\top$	$\perp$
$+$	$+$	0	$-$	$\top$	$\perp$
0	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
$-$	$-$	0	$+$	$\top$	$\perp$
$\top$	$\top$	0	$\top$	$\top$	$\perp$
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

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

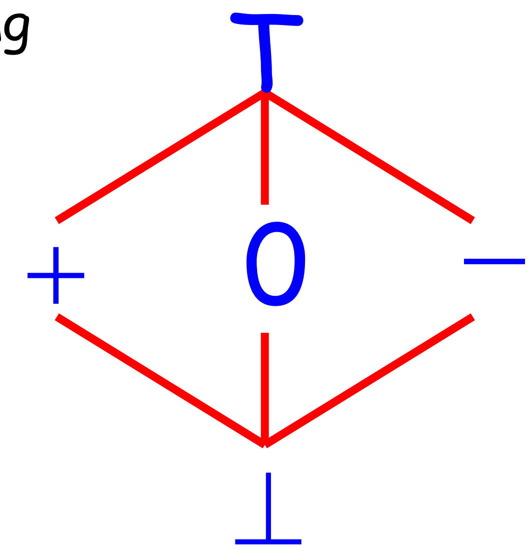
$$\bot \ \bar{+} \ x = \bot$$

$$x \ \bar{\times} \ \bot = \bot$$

$$\bar{-} \ \bot = \bot$$

The Abstract Domain

- Our abstract domain forms a *lattice*.
  - A partial order $x \le y \Leftrightarrow \gamma(x) \subseteq \gamma(y)$
  - 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.

$$\alpha : 2^{\mathrm{Int}} \to A$$

$$\alpha(S) = \mathrm{lub}\left(\{- \mid i < 0 \wedge i \in S\}, \{0 \mid 0 \in S\}, \{+ \mid i > 0 \wedge i \in S\}\right)$$

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

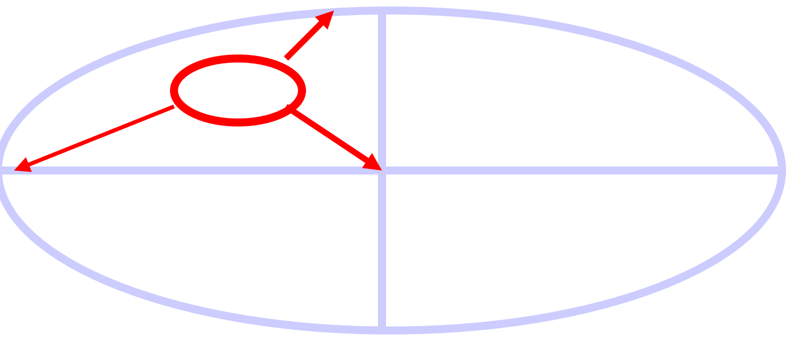
$$id \leq \gamma \circ \alpha$$
$$id = \alpha \circ \gamma$$

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.

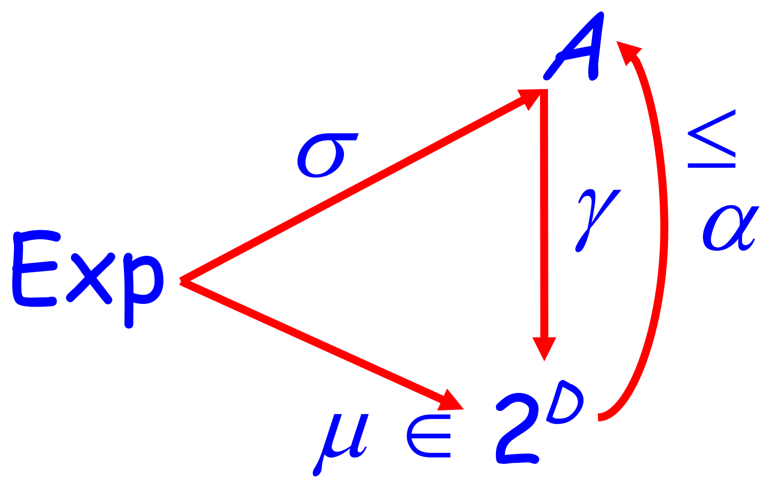
$$id \leq \gamma \circ \alpha$$

$$id = \alpha \circ \gamma$$



Picture

- In correct abstract interpretations, we expect the following diagram to commute.



General Conditions for Correctness

Three conditions guarantee correctness in general:

α and γ form a Galois insertion

$$id \leq \gamma \circ \alpha, \; id = \alpha \circ \gamma$$

α and γ are monotonic

$$x \leq y \Rightarrow \alpha(x) \leq \alpha(y)$$

Abstract operations $\overline{\mathrm{op}}$ are locally correct:

$$\gamma(\overline{\mathrm{op}}(s_1, \ldots, s_n)) \supseteq \mathrm{op}(\gamma(s_1), \ldots, \gamma(s_n))$$

Generic Correctness Proof

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

$$
\begin{array}{lll}
 & \mu(e_1 \ op \ e_2) & \\
= & \mu(e_1) \ op \ \mu(e_2) & \text{def. of } \mu \\
\in & \gamma(\sigma(e_1)) \ op \ \gamma(\sigma(e_2)) & \text{by induction} \\
\subseteq & \gamma(\sigma(e_1) \ \overline{op} \ \sigma(e_2)) & \text{local correctness} \\
= & \gamma(\sigma(e_1 \ op \ e_2)) & \text{def of } \sigma
\end{array}
$$

A Second Notion of Correctness

- We can define correctness using abstraction instead of concretization.

$$\mu(e) \in \gamma(\sigma(e)) \equiv \alpha(\{\mu(e)\}) \le \sigma(e)$$

$\Rightarrow$ direction

$$\mu(e) \in \gamma(\sigma(e))$$

$$\alpha(\{\mu(e)\}) \le \alpha(\gamma(\sigma(e))) \quad \text{monotonicity}$$

$$\alpha(\{\mu(e)\}) \le \sigma(e) \qquad \alpha \circ \gamma = id$$

Correctness (Cont.)

- The other direction . . .

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

$\Leftarrow$ direction

$$\alpha(\{\mu(e)\}) \leq \sigma(e)$$

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

$$\mu(e) \in \gamma(\sigma(e)) \quad id \leq \gamma \circ \alpha$$

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.

$$e = i \mid e * e \mid -e \mid \ldots \mid x$$

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.

$$\begin{aligned}
\mu_i(j) &= i \\
\mu_x(j) &= j \\
\mu_{e_1 * e_2}(j) &= \mu_{e_1}(j) * \mu_{e_2}(j) \\
\mu_{e_1 + e_2}(j) &= \mu_{e_1}(j) + \mu_{e_2}(j) \\
\ldots &= \ldots
\end{aligned}$$

Abstract Semantics

- Abstract semantic function:

$$\sigma : \text{Exp} \rightarrow A \rightarrow A$$

- Also write this semantics curried.

$$
\begin{aligned}
\sigma_i(\bar{j}) &= \bar{i} \\
\sigma_x(\bar{j}) &= \bar{j} \\
\sigma_{e_1 * e_2}(\bar{j}) &= \sigma_{e_1}(\bar{j}) \mathbin{\bar{*}} \sigma_{e_2}(\bar{j}) \\
\sigma_{e_1 + e_2}(\bar{j}) &= \sigma_{e_1}(\bar{j}) \mathbin{\bar{+}} \sigma_{e_2}(\bar{j}) \\
\ldots &= \ldots \\
\bar{i} &= \alpha(\{i\})
\end{aligned}
$$

Correctness

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

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

$$\mu_e \leq_D \gamma \circ \sigma_e \circ \alpha$$

$$\alpha \circ \mu_e \leq_A \sigma_e \circ \alpha$$

$$\begin{tikzcd}
A \arrow[r, "\sigma_e"] & A \arrow[d, "\gamma"] \\
2^D \arrow[u, "\alpha"] \arrow[r] & \mu_e \in 2^D \arrow[u, bend right=60, "\overset{\leq}{\alpha}"']
\end{tikzcd}$$

Local Correctness

- We also need a modified local correctness condition.

$$op\left(\gamma(\sigma_{e_1}(\bar{j})),\ldots,\gamma(\sigma_{e_n}(\bar{j}))\right) \subseteq \gamma\left(\overline{op}(\sigma_{e_1}(\bar{j}),\ldots,\sigma_{e_n}(\bar{j}))\right)$$

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

$$
\begin{array}{lll}
 & \mu_{op(e_1,\ldots,e_n)}(j) & \\
= & op(\mu_{e_1}(j),\ldots,\mu_{e_n}(j)) & \text{def. of } \mu \\
\in & op(\gamma(\sigma_{e_1}(\bar{j})),\ldots,\gamma(\sigma_{e_n}(\bar{j})) & \text{induction} \\
\subseteq & \gamma(\overline{op}(\sigma_{e_1}(\bar{j}),\ldots,\sigma_{e_n}(\bar{j}))) & \text{local correctness} \\
= & \gamma(\sigma_{op(e_1,\ldots,e_n)}(\bar{j})) & \text{def. of } \sigma
\end{array}
$$

If-Then-Else

$$e = \ldots \mid \text{if } e = e \text{ then } e \text{ else } e \mid \ldots$$

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

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

- 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.)

$$
\begin{array}{lll}
 & \mu_{e_3}(i) & \\
\in & \gamma(\sigma_{e_3}(\bar{i})) & \text{by induction} \\
\subseteq & \gamma(\sigma_{e_3}(\bar{i})) \sqcup \gamma(\sigma_{e_4}(\bar{i})) & \\
\subseteq & \gamma\left(\sigma_{e_3}(\bar{i}) \sqcup \sigma_{e_4}(\bar{i})\right) & \text{monotonicity of } \gamma
\end{array}
$$

Recursion

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

$$\mu : \text{Exp} \to \text{Int} \to \text{Int}_\perp$$

$$\textit{program} \; = \; \text{def } f(x) = e$$

$$e = \ldots \mid f(e)$$

Revised Meaning Function

- Define an auxiliary semantics taking a function (for the free variable *f*) and an integer (for *x*).

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

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

$$\mu'_x(g)(j) = j$$

$$\mu'_{e_1+e_2}(g)(j) = \mu'_{e_1}(g)(j) + \mu'_{e_2}(g)(j)$$

Meaning of Recursive Functions

$\mu : \text{Exp} \to \text{Int} \to \text{Int}_\perp$

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

Consider a function $\text{def f} = e$

Define an ascending chain $f_0, f_1, \ldots$ in $\text{Int} \to \text{Int}_\perp$

$f_0 = \lambda x. \perp$

$f_{i+1} = \mu'_e(f_i)$

Define $\mu_\text{f} = \bigcup_i f_i$

Abstract Semantics Revised

- Define an analogous auxiliary function for the abstract semantics.

$$\sigma' : \text{Exp} \to (A \to A) \to A \to A$$

$$\sigma'_{f(e)}(g)(\bar{i}) = g(\sigma'_e(g)(i))$$

$$\sigma'_x(g)(\bar{i}) = \bar{i}$$

$$\sigma'_{e_1+e_2}(g)(\bar{i}) = \sigma'_{e_1}(g)(\bar{i}) + \sigma'_{e_2}(g)(\bar{i})$$

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

$\sigma: \text{Exp} \to A \to A$

$\sigma': \text{Exp} \to (A \to A) \to A \to A$

Consider a function $\text{def f} = e$

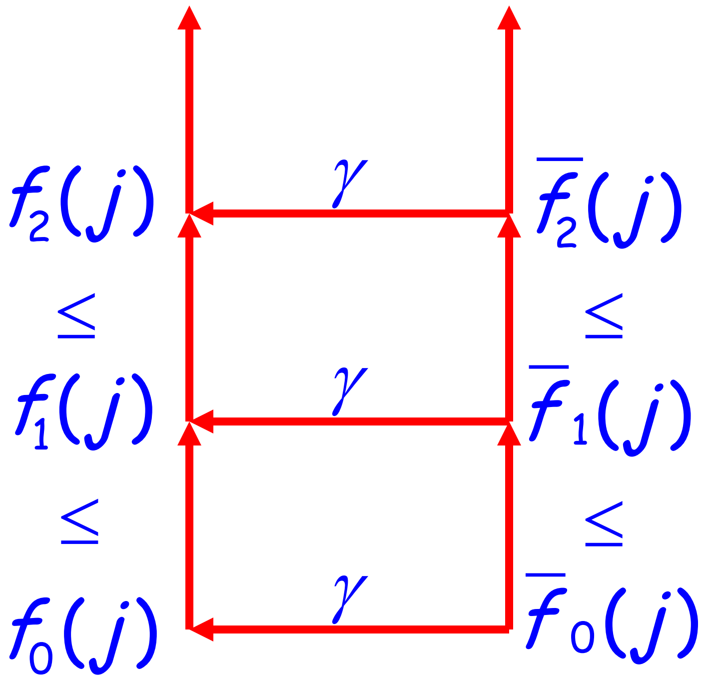
Define an ascending chain $\bar{f}_0, \bar{f}_1, \ldots$ in $A \to A$

$\bar{f}_0 = \lambda a.\perp$

$\bar{f}_{i+1} = \sigma'_e(\bar{f}_i)$

Define $\sigma_{\text{f}} = \bigcup_i \bar{f}_i$

Correctness



Corresponding elements of the chain stand in the correct relationship.

Correctness (Cont.)

$$\forall i.\ \ f_i(j) \in \gamma(\overline{f}_i(\overline{j}))$$

$$\Rightarrow \ \ \bigcup_{i \geq 0} f_i(j) \in \bigcup_{i \geq 0} \gamma(\overline{f}_i(\overline{j}))$$ chains stabilize

$$\Rightarrow \ \ \bigcup_{i \geq 0} f_i(j) \in \gamma\left(\bigcup_{i \geq 0} \overline{f}_i(\overline{j})\right)$$ monotonicity of γ

$$\Rightarrow \ \ \mu_f(j) \in \gamma(\sigma_f(\overline{j}))$$ by definition

Example

$$\mathsf{def}\ f(x) = \quad \mathsf{if}\ x = 0\ \mathsf{then}\ 1\ \mathsf{else}\ x * f(x + \text{-}1)$$

Abstraction:

$$\mathsf{lfp}\left(\sigma'\left(\mathsf{if}\ x = 0\ \mathsf{then}\ 1\ \mathsf{else}\ x * f(x + \text{-}1)\right)\right)$$

Simplified:

$$\mathsf{lfp}\left(\lambda \bar{f}.\lambda \bar{x}. + \cup \left(\bar{x}\ \ \bar{\times}\ \ \bar{f}(\bar{x}\ \ \bar{+}\ \ -)\right)\right)$$

Strictness

- We will assume our language is strict.
  - Makes little difference in quality of analysis for this example.
- Assume that $f(\perp)=\perp$
- Therefore it is sound to define $\bar{f}(\perp)=\perp$

Calculating the LFP

$$\mathrm{lfp}\left(\lambda \bar{f}.\lambda \bar{x}. + \cup \left(\bar{x} \ \bar{\times} \ \bar{f}(\bar{x} \ \bar{+} \ -)\right)\right)$$

$\bar{f}_0 =$

$\perp$	$-$	0	$+$	$\top$
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

$\bar{f}_1 =$

$\perp$	$-$	0	$+$	$\top$
$\perp$	$+$	$+$	$+$	$+$

$\bar{f}_2 =$

$\perp$	$-$	0	$+$	$\top$
$\perp$	$\top$	$\top$	$+$	$\top$

$\bar{f}_3 =$

$\perp$	$-$	0	$+$	$\top$
$\perp$	$\top$	$\top$	$\top$	$\top$

Notes

- In this case, the abstraction yields no useful information!
- Note that sequence of functions forms a *strictly* ascending chain until stabilization

$$f_0 < f_1 < f_2 < f_3 = f_4 = f_5 = \ldots$$

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

$$f_0(+) < f_1(+) = f_2(+) < f_3(+) = f_4(+) = f_5(+) = \ldots$$

Notes (Cont.)

- Lesson: The fixed point is being computed in the domain $(A \to A) \to A \to A$
- The fixed point is not being computed in $A \to A$
- 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 $f(\perp) = \perp$

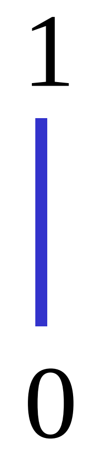
- Observation: if f is strict, then it is correct to pass arguments to f by value.

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

$$\gamma(0) = \{\perp\} \qquad \alpha(\{\perp\}) = 0$$

$$\gamma(1) = D \qquad \alpha(S) = 1 \text{ if } S \neq \{\perp\}$$

Abstract Semantics

- Next step is to define an abstract semantics

- Transform $f: Int \to Int$ to $\bar{f}: 2 \to 2$

- Transform values $v: Int$ 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.

$$
\begin{aligned}
\sigma'_x(g)(a) &= a \\
\sigma'_i(g)(a) &= 1 \\
\sigma'_{-e}(g)(a) &= \sigma'_e(g)(a) \\
\sigma'_{e_1 * e_2}(g)(a) &= \sigma'_{e_1}(g)(a) \wedge \sigma'_{e_2}(g)(a) \\
\sigma'_{f(e)}(g)(a) &= g(\sigma'_e(g)(a))
\end{aligned}
$$

The Rest of the Rules

$$
\begin{aligned}
\sigma'_{e_1+e_2}(g)(a) &= \sigma'_{e_1}(g)(a) \wedge \sigma'_{e_2}(g)(a) \\
\sigma'_{e_1/e_2}(g)(a) &= \sigma'_{e_1}(g)(a) \wedge \sigma'_{e_2}(g)(a) \\
\sigma'_{\text{if } e_1=e_2 \text{ then } e_3 \text{ else } e_4}(g)(a) &= \sigma'_{e_1}(g)(a) \wedge \sigma'_{e_2}(g)(a) \wedge \left(\sigma'_{e_3}(g)(a) \vee \sigma'_{e_4}(g)(a)\right) \\
\sigma_{\text{def f } = e} &= \text{lfp } \sigma'_e
\end{aligned}
$$

An Example

$$\text{def } f(x) \;=\; \text{if } x = 0 \text{ then } 1 \text{ else } x * f(x + \text{-}1)$$

$$\text{lfp}\left(\sigma'\left(\text{if } x = 0 \text{ then } 1 \text{ else } x * f(x + \text{-}1)\right)\right)$$

$$\text{lfp}\left(\lambda \bar{f}.\lambda \bar{x}.\bar{x}\right) = \lambda a.a$$

$(\lambda a.a)\; 0 = 0$ The function is strict in x.

Calculating the LFP

$$\mathrm{lfp}\left(\lambda\bar{f}.\lambda\bar{x}.\bar{x} \wedge 1 \wedge \left(1 \vee (\bar{x} \wedge \bar{f}(\bar{x} \wedge 1))\right)\right)$$

$$\bar{f}_0 = \begin{array}{|c|c|} \hline 0 & 1 \\ \hline 0 & 0 \\ \hline \end{array}$$

$$\bar{f}_1 = \begin{array}{|c|c|} \hline 0 & 1 \\ \hline 0 & 1 \\ \hline \end{array}$$

$$\bar{f}_2 = \begin{array}{|c|c|} \hline 0 & 1 \\ \hline 0 & 1 \\ \hline \end{array}$$

Another Example

- Generalize to recursive functions of two variables.

$$\text{def } f(x,y) = \text{ if } x = 0 \text{ then } 0 \text{ else } f(x + \text{-}1, f(x,y))$$

$$\text{lfp}\left(\sigma'\left(\text{if } x = 0 \text{ then } 0 \text{ else } f(x + \text{-}1, f(x,y))\right)\right) =$$

$$\text{lfp}(\lambda \bar{f}.\lambda(\bar{x},\bar{y}).\ \bar{x} \wedge 1 \wedge (1 \vee \ldots)) =$$

$$\lambda(\bar{x},\bar{y}).\ \bar{x}$$

Example (Cont.)

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

$$\left(\lambda(\bar{x},\bar{y}).\ \bar{x}\right)\ (0,1)=0$$ *x can be passed by value.*

$$\left(\lambda(\bar{x},\bar{y}).\ \bar{x}\right)\ (1,0)=1$$ *Unsafe to pass y by value.*

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 senstivity

- 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