Review and Today

- We introduced abstract interpretation
- An abstraction mapping from concrete to abstract values
  - Has a concretization mapping which forms a Galois connection
- We’ll look a bit more at Galois connections
- We’ll lift AI from expressions to programs
- ... and we’ll discuss the mythic "widening"

Three Little Correctness Conditions

- Three conditions define a correct abstract interpretation
  1. $\alpha$ and $\gamma$ are monotonic
  2. $\alpha$ and $\gamma$ form a Galois connection
     = "$\alpha$ and $\gamma$ are almost inverses"
  3. Abstraction of operations is correct
     $a_1 \oplus a_2 = \alpha(\gamma(a_1) \oplus \gamma(a_2))$

Why Galois Connections?

- We have an abstract domain $A$ (e.g., $\{-, 0, +\}$)
- Concrete domain $C$ (e.g., $\mathbb{Z}$)
- Abstraction function
  - $\alpha : \mathcal{P}(C) \rightarrow A$ or $\beta : C \rightarrow A$
- Concretization function
  - $\gamma : A \rightarrow \mathcal{P}(C)$
- Abstract semantics
  - $\sigma : \text{Exp} \rightarrow A$ (abstract operators $\oplus^A$)
- Concrete semantics
  - $\llbracket \cdot \rrbracket : \text{Exp} \rightarrow C$
Galois Connections

- A pair of functions between lattices $A$ and $\mathcal{P}(C)$
  - $\gamma$ and $\alpha$ are monotonic (with $\subseteq$ ordering on $\mathcal{P}(C)$)
  - $\alpha(\gamma(a)) = a$ for all $a \in A$
  - $\gamma(\alpha(S)) \subseteq S$ for all $S \in \mathcal{P}(C)$

More on Galois Connections

- All Galois connections are monotonic
- In a Galois connection one function uniquely and absolutely determines the other

End of Review

Questions?

Collecting Semantics

- Recall
  - A state $\sigma \in \Sigma$. Any state $\sigma$ has type $\text{Var} \rightarrow \mathbb{Z}$
  - States vary from program point to program point
- We introduce a set of program points: labels
- We want to answer questions like:
  - Is $x$ always positive at label $i$?
  - Is $x$ always greater or equal to $y$ at label $j$?

Abstract Interpretation for Imperative Programs

- So far we abstracted the value of expressions
- Now we want to abstract the state at each point in the program
- First we define the concrete semantics that we are abstracting
  - We'll use a collecting semantics

Collecting Semantics

- To answer these questions we'll construct $C \in \text{Contexts}$. $C$ has type $\text{Labels} \rightarrow \mathcal{P}(\Sigma)$
  - For each label $i$,
    - $C(i)$ = all possible states at label $i$
  - This is called the collecting semantics of the program
  - This is basically what SLAM (and BLAST, ESP, ...) approximate (using BDDs to store $\mathcal{P}(\Sigma)$ efficiently)
Defining the Collecting Semantics

- We first define relations between the collecting semantics at different labels
- We do it for unstructured CFGs (flowchart programs)
- Can do it for IMP with careful notion of program points
- Define a label on each edge in the CFG
- For assignment
  \[ x := e \]
  \[ C_j = \{ \sigma[x := n] \mid \sigma \in C_i \land [e][\sigma] = n \} \]
- For conditionals
  \[ \begin{array}{cccc}
  \text{false} & b & \text{true} \\
  \text{else} & & \text{then} \\
  \end{array} \]
  \[ C_{\text{else}} = \{ \sigma \mid \sigma \in C_{\text{in}} \land [b][\sigma] = \text{false} \} \]
  \[ C_{\text{then}} = \{ \sigma \mid \sigma \in C_{\text{in}} \land [b][\sigma] = \text{true} \} \]
- Assumes \( b \) has no side effects (as in IMP)

- For a join
  \[ C_{\text{out}} = C_i \cup C_j \]
- Consider the following program (assume \( x \geq 0 \) initially)
  \[ \begin{array}{cccc}
  y := 1 \\
  y := y \times x \\
  x := x - 1 \\
  \end{array} \]
Collecting Semantics: Example

- Consider the following program (assume \( x \geq 0 \) initially)

\[
\begin{align*}
\text{1:} & \quad y := 1 \\
\text{2:} & \quad x == 0 \\
\text{3:} & \quad y := y \times x \\
\text{4:} & \quad x := x - 1 \\
\text{5:} & \quad F \quad T
\end{align*}
\]

\[
\begin{align*}
C_1 &= \{ \sigma \mid \sigma(x) \geq 0 \} \\
C_2 &= \{ \sigma[y := 1] \mid \sigma \in C_3 \} \\
\cup \{ \sigma[x := \sigma(x) - 1] \mid \sigma \in C_4 \} \\
C_3 &= C_2 \cap \{ \sigma \mid \sigma(x) > 0 \} \\
C_4 &= C_2 \cap \{ \sigma \mid \sigma(x) = 0 \} \\
C_5 &= (\sigma[y := \sigma(y) \times \sigma(x)] \mid \sigma \in C_3)
\end{align*}
\]

Why Does This Work?

- We just made a system of recursive equations that are defined largely in terms of themselves
  - e.g., \( C_2 = F(C_4) \), \( C_4 = G(C_3) \), \( C_3 = H(C_2) \)
- Why do we have any reason to believe that this will get us what we want?

The Collecting Semantics

- We have an equation with the unknown \( C \)
  - The equation is defined by a monotonic and continuous function on the domain \( \text{Labels} \rightarrow \mathcal{P}(\Sigma) \)
- We can use the least fixed-point theorem
  - Start with \( C^0(\{\}) = 0 \) (aka \( C^0 = \lambda \mathcal{L}.0) \)
  - Apply the relations between \( C_i \) and \( C_j \) to get \( C_i \) from \( C_{i-1} \)
  - Stop when all \( C^k = C^{k-1} \)
  - Problem: we'll go on forever for most programs
  - But we know the fixed point exists
Consider the following program (assume $x \geq 0$ initially):

1. $y := 1$ ($x \geq 0, y = 1$)
2. $x := x - 1$ ($x = 0$)
3. $y := y \times x$ ($x \geq 0, y = x$)
4. $x := x - 1$ ($x \geq 0, y = x$)

Collecting Semantics: Example

- Consider the following program (assume $x \geq 0$ initially):
  1. $y := 1$ ($x \geq 0, y = 1$)
  2. $x := x - 1$ ($x = 0$)
  3. $y := y \times x$ ($x \geq 0, y = x$)
  4. $x := x - 1$ ($x \geq 0, y = x$)

Collecting Semantics: Example

- Consider the following program (assume $x \geq 0$ initially):
  1. $y := 1$ ($x \geq 0, y = 1$)
  2. $x := x - 1$ ($x = 0$)
  3. $y := y \times x$ ($x \geq 0, y = x$)
  4. $x := x - 1$ ($x \geq 0, y = x$)

Collecting Semantics: Example

- Consider the following program (assume $x \geq 0$ initially):
  1. $y := 1$ ($x \geq 0, y = 1$)
  2. $x := x - 1$ ($x = 0$)
  3. $y := y \times x$ ($x \geq 0, y = x$)
  4. $x := x - 1$ ($x \geq 0, y = x$)
Abstract Interpretation on CFG

- Pick a complete lattice \( A \) (abstractions for \( P(\Sigma) \))
  - Along with a monotonic abstraction \( \alpha : \Sigma \rightarrow A \)
  - Alternatively, pick \( \beta : \Sigma \rightarrow A \)
  - This uniquely defines its Galois connection \( \gamma \)
- Take the relations between \( C_i \) and move them to the abstract domain:
  \[ a : Label \rightarrow A \]
- Assignment
  \[ C_j = \{ \sigma | \sigma \in C_i \wedge [e] \sigma = n \} \]
  \[ a_j = \alpha \{ \sigma | \sigma \in \gamma (a_i) \wedge [e] \sigma = n \} \]
- Conditional
  \[ C_j = \{ \sigma | \sigma \in C_i \wedge [b] \sigma = \text{false} \} \]
  \[ a_j = \alpha \{ \sigma | \sigma \in \gamma (a_i) \wedge [b] \sigma = \text{false} \} \]
  \[ C_k = \{ \sigma | \sigma \in C_i \wedge [b] \sigma = \text{true} \} \]
  \[ a_k = \alpha \{ \sigma | \sigma \in \gamma (a_i) \wedge [b] \sigma = \text{true} \} \]
- Join
  \[ C_k = C_i \cup C_j \]
  \[ a_k = \alpha (\gamma (a_i) \cup \gamma (a_j)) = \text{lub} \{a_i, a_j\} \]

Least Fixed Points in the Abstract Domain

- We have a recursive equation with unknown "a"
  - Defined by a monotonic and continuous function on the domain \( Labels \rightarrow A \)
- We can use the least fixed-point theorem:
  - Start with \( a_0 = \lambda L. \bot \) (aka: \( a_0(L) = \bot \))
  - Apply the monotonic func to compute \( a^{k+1} \) from \( a^k \)
  - Stop when \( a^{k+1} = a^k \)
- Exactly the same computation as for the collecting semantics
  - What is new?

What’s new?

HOPE

Least Fixed Points in the Abstract Domain

- We have a hope of termination!
- Classic setup: \( A \) has only uninteresting chains (finite number of elements in each chain)
  - \( A \) has finite height \( h \) (= “finite-height lattice”)
- The computation takes \( O(h \times |Labels|^2) \) steps
  - At each step "a" makes progress on at least one label
  - We can only make progress \( h \) times
  - And each time we must compute \( |Labels| \) elements
- This is a quadratic analysis: good news
  - This is exactly the same as Kildall’s 1973 analysis of dataflow’s polynomial termination given a finite-height lattice and monotonic transfer functions.

That’s It!

Program Analysis in a Nutshell

Define an Abstraction

Compute a Fixed Point in the Abstraction
Abstract Interpretation: Example

- Consider the following program

\[
\begin{align*}
  &y := 1 \\
  &x == 0 \\
  &y := y \times x \\
  &x := x - 1
\end{align*}
\]

We want to do sign analysis on it

Abstract Domain for Sign Analysis

- Invent the complete sign lattice
  \[ S = \{ \bot, -, 0, +, \top \} \]
- Construct the complete lattice
  \[ A = \{ x, y \} \to S \]

- With the usual point-wise ordering
- Abstract state gives the sign for \( x \) and \( y \)
- We start with \( a^0 = \lambda L.\lambda v \in \{ x, y \}. \bot \)
- aka: \( a^0(L,v) = \bot \)

Let's Do It!

<table>
<thead>
<tr>
<th>Label</th>
<th>Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x +</td>
</tr>
<tr>
<td></td>
<td>y T</td>
</tr>
<tr>
<td>2</td>
<td>x T +</td>
</tr>
<tr>
<td></td>
<td>y +</td>
</tr>
<tr>
<td>3</td>
<td>x \bot</td>
</tr>
<tr>
<td></td>
<td>y +</td>
</tr>
<tr>
<td>4</td>
<td>x +</td>
</tr>
<tr>
<td></td>
<td>y +</td>
</tr>
<tr>
<td>5</td>
<td>x \bot</td>
</tr>
<tr>
<td></td>
<td>y +</td>
</tr>
</tbody>
</table>

Let's Do It!

<table>
<thead>
<tr>
<th>Label</th>
<th>Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x +</td>
</tr>
<tr>
<td></td>
<td>y T</td>
</tr>
<tr>
<td>2</td>
<td>x T +</td>
</tr>
<tr>
<td></td>
<td>y +</td>
</tr>
<tr>
<td>3</td>
<td>x \bot</td>
</tr>
<tr>
<td></td>
<td>y +</td>
</tr>
<tr>
<td>4</td>
<td>x +</td>
</tr>
<tr>
<td></td>
<td>y +</td>
</tr>
<tr>
<td>5</td>
<td>x \bot</td>
</tr>
<tr>
<td></td>
<td>y +</td>
</tr>
</tbody>
</table>

Weaknesses

- We abstracted the state of each variable independently
  \[ A = \{ x, y \} \to \{ \bot, -, 0, +, \top \} \]
  - Our abstraction is non-relational
- We lost relationships between variables
  - e.g., at a point \( x \) and \( y \) may always have the same sign
  - In the previous abstraction, we get \( \{ x := \top, y := \top \} \) at label 2 (when in fact \( y \) is always positive!)

Potential Solutions?

- Can also abstract the state as a whole
  \[ A = \mathcal{P}(\{ \bot, -, 0, +, \top \} \times \{ \bot, -, 0, +, \top \}) \]
- For the previous example we now get the abstraction \( \{ (-, -), (0, 0), (+, +) \} \) at 2
Other Abstract Domains

- **Range analysis**
  - Lattice of ranges: \( R = \{ (-\infty, n], [n, m], (n, +\infty), \infty \} \)
  - It is a complete lattice
  - \([n..m] \uplus [n'..m'] = [\min(n, n')..\max(m, m')]\)
  - With appropriate care in dealing with \(+\infty\):
    - \(\beta: \mathbb{Z} \to R\) such that \(\beta(n) = [n..n]\)
    - \(\alpha: P(S) \to R\) such that \(\alpha(S) = \text{lub}\{\beta(n) | n \in S\} = [\min(S)..\max(S)]\)
    - \(\gamma: R \to P(\mathbb{Z})\) such that \(\gamma(r) = \{n | n \in r\}\)
  - This lattice has infinite-height chains
    - So the abstract interpretation might not terminate!

Example of Non-Termination

- Consider this (common) program fragment
  
  ```
  i := 0
  i <= n
  i := i + 1
  ```

  We want to do range analysis for it

  Consider the sequence of abstract states at point 2
  - \([1..1], [1..2], [1..3], \ldots\)
  - The analysis never terminates
  - Or terminates very late if the loop bound known statically

  It is time to approximate even more: widening

  We redefine the join (lub) operator of the lattice to ensure that from \([1..1]\) upon union with \([2..2]\) the result is \([1..+\infty]\) and not \([1..2]\)

  Now the sequence of states is
  - \([1..1], [1..\infty], [1..\infty] \ldots\) (no more infinite chains)

Uses of Widening

- Two different main uses:
  - Approximate missing lubs. (Not for us.)
  - Convergence acceleration. (This is the real use.)

The magic of program analysis

Example of Non-Termination

- Consider the sequence of abstract states at point 2
  - \([1..1], [1..2], [1..3], \ldots\)
  - The analysis never terminates
  - Or terminates very late if the loop bound known statically

It is time to approximate even more: widening

We redefine the join (lub) operator of the lattice to ensure that from \([1..1]\) upon union with \([2..2]\) the result is \([1..+\infty]\) and not \([1..2]\)

Now the sequence of states is
- \([1..1], [1..\infty], [1..\infty] \ldots\) (no more infinite chains)

Formal Definition of Widening

- A widening \(\triangledown: (P \times P) \to P\) on a poset \((P, \subseteq)\) satisfies:
  - For all \(x, y \in P\) \(x \triangledown y \subseteq \text{upper bound condition}\)
  - For all increasing chains \(x^0 \subseteq x^1 \subseteq \ldots\)
    - the increasing chain \(y^0 \triangledown x^0, y^1 \triangledown x^1, \ldots\)
      - is not strictly increasing.

Formal Widening Example

\([1,1] \triangledown [1,2] = [1, +\infty]\)

- Range Analysis on \(z\):
  
<table>
<thead>
<tr>
<th>Original (x^i)</th>
<th>Widened (y^i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x^{0}_{L0} = T)</td>
<td>(y^{0}_{L0} = T)</td>
</tr>
<tr>
<td>(x^{1}_{L1} = [1,1])</td>
<td>(y^{1}_{L1} = [1,1])</td>
</tr>
<tr>
<td>(x^{2}_{L2} = [1,1])</td>
<td>(y^{2}_{L2} = [1,1])</td>
</tr>
<tr>
<td>(x^{3}_{L3} = [2,2])</td>
<td>(y^{3}_{L3} = [2,2])</td>
</tr>
<tr>
<td>(x^{4}_{L4} = [1,2])</td>
<td>(y^{4}_{L4} = [1, +\infty])</td>
</tr>
</tbody>
</table>

Where \(x^i_{Li}\) is the \(i\)th iterative attempt to compute an abstract value for \(z\) at label \(Li\)

Recall lub \(S = [\min(S), \max(S)]\)

\(\text{lub}([2..3], [1..1]) = [1..3]\)

8
Other Abstract Domains

- Linear relationships between variables
  - A convex polyhedron is a subset of \( \mathbb{Z}^k \) whose elements satisfy a number of inequalities:
    \[ a_1 x_1 + a_2 x_2 + \cdots + a_k x_k \geq c \]
  - This is a complete lattice; linear programming methods compute lubs
- Linear relationships with at most two variables
  - Convex polyhedra but with \( \leq 2 \) variables per constraint
  - Octagons \((x + y \geq c)\) have efficient algorithms
- Modulus constraints (e.g. even and odd)

Abstract Chatter

- AI, Dataflow, and Software Model Checking
  - The big three (aside from flow-insensitive type systems) for program analyses
  - Are in fact quite related:
    - David Schmidt. *Data flow analysis is model checking of abstract interpretation*, POPL ’98.
  - AI is usually flow-sensitive (per-label answer)
  - AI can be path-sensitive (if your abstract domain includes \( \lor \), for example), which is just where model checking uses BDD’s
  - Metal, SLAM, ESP, … can all be viewed as AI

Abstract Interpretation Summary

- AI is a very powerful technique that underlies a large number of program analyses
- AI can also be applied to functional and logic programming languages
- When the lattices have infinite height and widening heuristics are used, the result become harder to predictable
- AI is behind Astrée, which is used by Airbus