Interprocedural Analysis and Abstract Interpretation
Outline

- Interprocedural analysis
  - control-flow graph
  - MVP: “Meet” over Valid Paths
  - Making context explicit
    - Context based on call-strings
    - Context based on assumption sets

- Abstract interpretation
Control-flow graph for a whole program

- At each function definition proc $p(x)$
  - Create two special CFG nodes:
    - init($p$) and final($p$)
  - Build CFG for the function body
    - Use init($p$) as the function entry node
    - Connect every return node to final($p$)

- At each function call to $p(x)$ with
  - Split the original function call into two stmts
    - Enter $p(x)$ (before making the call) and exit $p(x)$ (after the call exits)
  - Connect enter $p(x)$ -> init($p$), final($p$) -> exit $p(x)$
  - Connect enter $p(x)$ -> exit $p(x)$ to allow the flow of extra context info

- Three kinds of CFG edges
  - Intra-procedural: internal control-flow within a procedure
  - Procedure calls: from enter $p(x)$ to init($p$)
  - Procedure returns: from final($p$) to exit $p(x)$
Interprocedural CFG Example

Example:

```c
int fib(int z) {
    if (z < 3) then return 1;
    else return fib(z-1) + fib(z-2);
}
```

Main program: return fib(15);

```c
t = exit fib(15)
```

Problem: matching between function calls and returns
Extending monotone frameworks

- Monotone frameworks consists of
  - A complete lattice \((L, \leq)\) that satisfies the Ascending Chain Condition
  - A set \(F\) of monotone transfer functions from \(L\) to \(L\) that
    - contains the identity function and
    - is closed under function composition

- Transfer functions for procedure definitions
  - For simplicity, both \(\text{init}(p)\) and \(\text{final}(p)\) have identity transfer functions

- Transfer functions for procedure calls
  - For procedure entry: assign values to formal parameters
  - For procedure exit: assign return values to outside
Problem: calling context upon return

int fib(int z) {
    if (z < 3) then return 1;
    else return fib(z-1) + fib(z-2);
}
Main program: return fib(15);

- Matching between function calls and returns
  - Calculating solutions on non-existing paths could seriously
detriment precision
    - E.g. enter fib(z-2) -> init(fib) -> ... -> exit fib(z-1) -> ...

- 
- Matching between function calls and returns
- Calculating solutions on non-existing paths could seriously
detriment precision
- E.g. enter fib(z-2) -> init(fib) -> ... -> exit fib(z-1) -> ...

B0: init(fib)
B1: if (z < 3)
B2: enter fib(z-1)
B3: t1 = exit fib(z-1) enter fib(z-2)
B4: t2 = exit fib(z-2) return t1 + t2;
B5: return 1
B6: final(fib)
MVP: “Meet” over Valid Paths

- Problem: matching procedure entries and exits (function calls and returns)
- A complete path must
  - Have proper nesting of procedure entries and exits
  - A procedure always return to the point immediately after it is called
- A valid path must
  - Start at the entry node of the main program
  - All the procedure exits match the corresponding entries
  - Some procedures may be entered but not yet exited
- The MVP solution
  - At each program point $t$, the solution for $t$ is
    - $\text{MVP}(t) = \land \{ \text{sol}(p) : p \text{ is a valid path to } t \}$
Making Context Explicit

- **Context sensitive analysis**
  - Maintain separate solutions for different callers of a function

- **Extending the monotone framework**
  - Starting point (context-insensitive)
    - A complete lattice \((L, \leq)\) that satisfies the Ascending Chain Condition
      - \(L = \text{Power}(D)\) where \(D\) is the domain of each solution
    - A set \(F\) of monotone transfer functions from \(L\) to \(L\)
  - Extension
    - \(L = \text{Power}(D \ast C)\), where \(C\) includes all calling contexts
    - \(F = L \rightarrow L\), a separate sub-solution is calculated for each calling context
      - \(F\) (procedure entry): attach caller info. to incoming solution
      - \(F\) (procedure exit): match caller info, eliminate solution for invalid paths
Different Kinds of Context

- **Call strings --- contexts based on control flow**
  - Remember a list of procedure calls leading to the current program point
    - Call strings of unbounded length --- remember all the preceding calls
    - Call strings of bounded length (k) --- remember only the last k calls

- **Assumption sets --- contexts based on data flow**
  - Assumption sets
    - Use the solution before entering proc p(x) as calling context (e.g., each context makes distinct presumptions about values of function parameters)
  - Large vs. small assumption sets
    - How large is the context: use the entire solution or pick a single constraint from the solution
Example Context-sensitive Analysis

Range analysis: for each variable reference $x$, is its value $\geq$ or $\leq$ a constant value? (i.e., $x \geq x_1; z \leq n_2$)?
# Example Range Analysis

Variables: $x, z, t_1, t_2, \text{fib, } t$; Contexts: A0, B2, B3, none; Domain: Variables * ($\leq n, = n, \geq n, ?, \text{any}$)

<table>
<thead>
<tr>
<th></th>
<th>A0</th>
<th>B0</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
<th>B4</th>
<th>B5</th>
<th>B6</th>
<th>A1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Contexts:</strong></td>
<td>(none)</td>
<td>(none, z=?(A0,z=15) (B2/B3, z=?)</td>
<td>(none)</td>
<td>(none)</td>
<td>(none)</td>
<td>(none)</td>
<td>(none, z=?(A0,z=15) (B2/B3, z=?)</td>
<td>(none)</td>
<td>(none)</td>
</tr>
<tr>
<td><strong>Domain:</strong></td>
<td>Variables * ($\leq n, = n, \geq n, ?, \text{any}$)</td>
<td>(B0,z=15) (B2/B3, z=?)</td>
<td>(A0,z=15)(B2/z&gt;=2) (B3/z&gt;=1)</td>
<td>(A0,z=15)(B2/z&gt;=2) (B3/z&gt;=1)</td>
<td>(A0,z=15)(B2/z&gt;=2) (B3/z&gt;=1)</td>
<td>(A0,z=15)(B2/z&gt;=2) (B3/z&gt;=1)</td>
<td>(A0,z=15)(B2/B3,z&gt;=3)</td>
<td>(A0,z=15) (B2/z&gt;=3) (B3/z&gt;=1)</td>
<td>(none)</td>
</tr>
<tr>
<td><strong>A0</strong></td>
<td>(none)</td>
<td>(none)</td>
<td>(none)</td>
<td>(none)</td>
<td>(none)</td>
<td>(none)</td>
<td>(none, z=?(A0,z=15) (B2/B3, z=?)</td>
<td>(none)</td>
<td>(none)</td>
</tr>
<tr>
<td><strong>B0</strong></td>
<td>(none, z=?)</td>
<td>(A0,z=15) (B2/B3, z=?)</td>
<td>(A0,z=15)(B2/z&gt;=2) (B3/z&gt;=1)</td>
<td>(A0,z=15)(B2/z&gt;=2) (B3/z&gt;=1)</td>
<td>(A0,z=15)(B2/z&gt;=2) (B3/z&gt;=1)</td>
<td>(A0,z=15)(B2/z&gt;=2) (B3/z&gt;=1)</td>
<td>(A0,z=15)(B2/B3,z&gt;=3)</td>
<td>(A0,z=15) (B2/z&gt;=3) (B3/z&gt;=1)</td>
<td>(none)</td>
</tr>
<tr>
<td><strong>B1</strong></td>
<td>(none, z=?)</td>
<td>(A0,z=15) (B2/B3, z=?)</td>
<td>(A0,z=15)(B2/z&gt;=2) (B3/z&gt;=1)</td>
<td>(A0,z=15)(B2/z&gt;=2) (B3/z&gt;=1)</td>
<td>(A0,z=15)(B2/z&gt;=2) (B3/z&gt;=1)</td>
<td>(A0,z=15)(B2/z&gt;=2) (B3/z&gt;=1)</td>
<td>(A0,z=15)(B2/B3,z&gt;=3)</td>
<td>(A0,z=15) (B2/z&gt;=3) (B3/z&gt;=1)</td>
<td>(none)</td>
</tr>
<tr>
<td><strong>B2</strong></td>
<td>(none, z=?)</td>
<td>(A0,z=15) (B2/B3, z=?)</td>
<td>(A0,z=15)(B2/B3,z&gt;=3)</td>
<td>(A0,z=15)(B2/B3,z&gt;=3)</td>
<td>(A0,z=15)(B2/B3,z&gt;=3)</td>
<td>(A0,z=15)(B2/B3,z&gt;=3)</td>
<td>(A0,z=15)(B2/B3,z&gt;=3)</td>
<td>(A0,z=15) (B2/z&gt;=3) (B3/z&gt;=1)</td>
<td>(none)</td>
</tr>
<tr>
<td><strong>B3</strong></td>
<td>(none, z=t1/?(A0,z=15,t1=?) (B2/B3,z&gt;=3,t1=?)</td>
<td>(A0,z=15)(B2/z&gt;=3) (B3,z1=t1=1)</td>
<td>(A0,z=15)(B2/z&gt;=3) (B3,z1=t1=1)</td>
<td>(A0,z=15)(B2/z&gt;=3) (B3,z1=t1=1)</td>
<td>(A0,z=15)(B2/z&gt;=3) (B3,z1=t1=1)</td>
<td>(A0,z=15)(B2/z&gt;=3) (B3,z1=t1=1)</td>
<td>(A0,z=15)(B2/B3,z&gt;=3)</td>
<td>(A0,z=15) (B2/z&gt;=3) (B3/z&gt;=1)</td>
<td>(none)</td>
</tr>
<tr>
<td><strong>B4</strong></td>
<td>(none, z/t1/t2=?)</td>
<td>(A0,z=15)(t1/t2=?) (B2/B3,z&gt;=3)</td>
<td>(A0,z=15)(t1/t2=1) (B2/B3,z&gt;=3)</td>
<td>(A0,z=15)(t1/t2=1) (B2/B3,z&gt;=3)</td>
<td>(A0,z=15)(t1/t2=1) (B2/B3,z&gt;=3)</td>
<td>(A0,z=15)(t1/t2=1) (B2/B3,z&gt;=3)</td>
<td>(A0,z=15)(B2/B3,z&gt;=3)</td>
<td>(A0,z=15) (B2/z&gt;=3) (B3/z&gt;=1)</td>
<td>(none)</td>
</tr>
<tr>
<td><strong>B5</strong></td>
<td>(none, z=?)</td>
<td>(B2/B3,z&lt;=?2)</td>
<td>(B2,z=2) (B3,z&lt;=?2)</td>
<td>(B2,z=2) (B3,z&lt;=?2)</td>
<td>(B2,z=2) (B3,z&lt;=?2)</td>
<td>(B2,z=2) (B3,z&lt;=?2)</td>
<td>(B2,z=2) (B3,z&lt;=?2)</td>
<td>(A0,z=15)(B2/z&gt;=3)</td>
<td>(A0,z=15) (B2/z&gt;=3)</td>
</tr>
<tr>
<td><strong>B6</strong></td>
<td>(none,z/fib=?)</td>
<td>(A0,z=15,fib=?) (B2/B3,z=any,fib=1)</td>
<td>(A0,z=15,fib=1) (B2/B3,z=any,fib=1)</td>
<td>(A0,z=15,fib=1) (B2/B3,z=any,fib=1)</td>
<td>(A0,z=15,fib=1) (B2/B3,z=any,fib=1)</td>
<td>(A0,z=15,fib=1) (B2/B3,z=any,fib=1)</td>
<td>(A0,z=15,fib=1) (B2/B3,z=any,fib=1)</td>
<td>(A0,z=15) (B2/z&gt;=3)</td>
<td>(A0,z=15) (B2/z&gt;=3)</td>
</tr>
<tr>
<td><strong>A1</strong></td>
<td>(none,t=?)</td>
<td>(none,t=?)</td>
<td>(none,t&gt;=1)</td>
<td>(none,t&gt;=1)</td>
<td>(none,t&gt;=1)</td>
<td>(none,t&gt;=1)</td>
<td>(none,t&gt;=1)</td>
<td>(none,t&gt;=1)</td>
<td>(none,t&gt;=1)</td>
</tr>
</tbody>
</table>
Foundations of Abstract Interpretation

Definition from Wikipedia

- **abstract interpretation** is a theory of sound approximation of the semantics of computer programs. It can be viewed as a partial execution of a computer program without performing all the calculations.

Outline

- Monotone frameworks
  - A complete lattice \((L,\leq)\) that satisfies the Ascending Chain Condition
  - A set \(F\) of monotone transfer functions from \(L\) to \(L\) that
    - contains the identity function and
    - is closed under function composition

- Galois connections, closures, and Moore families

- Soundness and completeness of operations on abstract data

- Soundness and completeness of execution trace computation
Galois Connections

- Two complete lattices
  - C: the “concrete” (execution) data
    - The execution of the entire program
    - Infinite and impossible to model precisely
  - A: the “abstract” (execution) data
    - Properties (abstractions) of the “concrete” data
    - The solution space (domain) of static program analysis

- For complete lattices C and A, a Galois connection is
  - A pair of monotonic functions, $\alpha : C \rightarrow A$, $\gamma : A \rightarrow C$
  - For all $a \in A$ and $c \in C$: $c \leq \gamma(\alpha(c))$ and $\alpha(\gamma(a)) \leq a$
  - Is Written as $C<\alpha,\gamma>A$
\textbf{Galois Connections (2)}

- $\gamma$ and $\alpha$ are inverse maps of each other's image
  - For all $c \in \gamma(A), c = \gamma(\alpha(c))$; for all $a \in \alpha(C), a = \alpha(\gamma(a))$
  - The maps $\alpha$ are "homomorphism" mappings between $C$ and $A$
- Galois connections are closed under
  - Composition, product, and so on
- Each instruction performs an action $f: C \rightarrow C$
  - Can use $\alpha$ and $\gamma$ to define an abstract transfer function $f^\# : A \rightarrow A$ for each $f: C \rightarrow C$
Closure Maps

- For $C<\alpha,\gamma> A$, it is common that $A \subseteq C$. This means $A$ embeds into $C$ as a sub-lattice
  - $A$’s elements name distinguished sets in $C$
- A closure map defines the embedding of $A$ within $C$.

**Definition:** $\rho: C \rightarrow C$ is a closure map if it is

- **Monotonic:** $\forall c1, c2 \in C, c1 \leq c2 \Rightarrow \rho(c1) \leq \rho(c2)$;
- **extensive:** $\forall c \in C, c \leq \rho(c)$;
- **idempotent:** $\forall c \in C, \rho(\rho(c)) = \rho(c)$ (i.e. $\rho \circ \rho = \rho$)

1) Every Galois connection, $C<\alpha,\gamma> A$ defines a closure map $\alpha \cdot \gamma$;
2) Every closure map, $\rho: C \rightarrow C$, defines the Galois connection, $C<\rho, id> \rho(C)$. 

Moore Families

- Given C, can we define a closure map on it by choosing some elements of C?
  - Yes, if the elements we select are closed under greatest-lower-bounds (meet) operation
  - That is, the new set of elements forms a complete lattice
- **Definition:** M ⊆ C is a *Moore family* iff for all S ⊆ M, (^S) ∈ M.
  - We can define a closure map as ρ(c)={c’ ∈ M | c ≤ c’}.
  - That is, we map each element in C to the closest abstraction (approximation) in M
- For each closure map, ρ:C->C, its image, ρ(C), is a Moore family.

Given C, we can define an abstract interpretation by selecting some M ⊆ C that is a Moore family
Closed Binary Relations

- Often the solution of an analysis is a power set of its domain
  - The Galois connection can be written as $\text{Power}(D) < \alpha, \gamma > A$
- Given unordered set $D$ and complete lattice $A$, it is natural to relate the elements in $D$ to those in $A$ by a binary relation, $R \subseteq D \times A$, s.t.
  - $(d,a) \in R$ (or $d \ R a$, $d \models R a$) means “$d$ has property $a$”.
  - Example: $D=\text{Int}$, $A=\{\text{none, neg, pos, zero, nonneg, nonpos, any}\}$.
    - Then $2 \ R \text{nonneg}$, $2 \ R \text{pos}$, and $2 \ R \text{any}$.
- The adjoint function, $\gamma : A \rightarrow \text{Power}(D)$, can be defined as
  - $\gamma(a) = \{d \in D \mid d \ R a\}$. E.g., $\gamma(\text{nonneg})=\{0,1,2,\ldots\}$.
  - If $R$ defines a Galois collection, then $\gamma(A)$ defines a Moore family.
- **Proposition:** $R \subseteq D \times A$ defines a Galois connection between $(\text{Power}(D), A)$ iff
  - $R$ is $U$-closed: $c \ R a$ and $a \leq a'$ imply $c \ R a'$;
  - $R$ is $G$-closed: $c \ R \{a \mid c \ R a\}$
Concrete and Abstract Operations

Now that we know how to model a solution space via abstraction function $\alpha : C \rightarrow A$,
- We must model concrete computation steps, $f : C \rightarrow C$, by abstract computation steps, $f^# : A \rightarrow A$.

Example: we have concrete domain, Nat, and concrete operation, $\text{succ} : \text{Nat} \rightarrow \text{Nat}$, defined as $\text{succ}(n) = n + 1$.
- abstract domain, Parity = {any, even, odd, none}.
- abstract operation, $\text{succ}^# : \text{Parity} \rightarrow \text{Parity}$, defined as
  - $\text{succ}^#(\text{even}) = \text{odd}$,
  - $\text{succ}^#(\text{odd}) = \text{even}$,
  - $\text{succ}^#(\text{any}) = \text{any}$,
  - $\text{succ}^#(\text{none}) = \text{none}$,
- $\text{succ}^#$ must be consistent (sound) with respect to $\text{succ}$:
  - if $n \text{ Rn a}$, then $\text{succ}(n) \text{ Rn succ}^#(a)$,
- where $\text{Rn} \subseteq \text{Nat} \times \text{Parity}$ relates numbers to their parities (e.g., 2 Rn even, 5 Rn odd, etc.).
Sound Approximation

- Given
  - Galois connection $C<\alpha, \gamma> A$ and
  - functions $f : C \rightarrow C$ and $f^# : A \rightarrow A$,

  $f^#$ is a *sound approximation* of $f$ iff
  - For all $c \in C$, $\alpha(f(c)) \leq f^#(\alpha(c))$
  - For all $a \in A$, $f(\gamma(a)) \leq \gamma(f^#(a))$

- That is, $\alpha$ defines a “semi-homomorphism” with respect to $f$ and $f^#$

\[ c \xrightarrow{\alpha} \alpha(c) \]
\[ \downarrow f \quad \downarrow f^# \]
\[ f(c) \xrightarrow{\alpha} \alpha(f(c)) \leq f^#(\alpha(c)) \]
Sound Approximation Example

- Given
  - Galois connection \( \text{Power}(\text{Nat}) \prec \alpha, \gamma \succ \text{Parity} \) and
  - Concrete transfer function \( \text{succ} : \text{Nat} \rightarrow \text{Nat}, \quad \text{succ}(S) = \{ n + 1 \mid n \in S \} \)
  - Abstract transfer function \( \text{succ}# : \text{Parity} \rightarrow \text{Parity}, \)
    \[ \text{succ}#(\text{even}) = \text{odd}, \quad \text{succ}#(\text{odd}) = \text{even} \]
    \[ \text{succ}#(\text{any}) = \text{any}, \quad \text{succ}#(\text{none}) = \text{none} \]
- \( \text{succ}# \) is a sound approximation of \( \text{succ} \)
  - For all \( c \in \text{Nat}, \quad \alpha(\text{succ}(c)) = \text{succ}#(\alpha(c)) \)

\[
\begin{array}{c}
\{2,6\} \\
\downarrow \text{succ} \\
\{3,7\}
\end{array}
\xrightarrow{\alpha}
\begin{array}{c}
\text{even} \\
\downarrow \text{succ}# \\
\text{odd}
\end{array}
\]
Synthesizing f# from f

- Given $C^{<\alpha,\gamma>}A$, and function $f : C \rightarrow C$, the most precise $f# : A \rightarrow A$ that is sound with respect to $f$ is
  - $f#_{\text{best}}(a) = \alpha(f(\gamma(a)))$

- **Proposition:** $f#$ is sound with respect to $f$ iff
  - For all $a \in A$, $f#_{\text{best}}(a) \leq f#(a)$
  - Of course, $f#_{\text{best}}$ has a *mathematical* definition—not an algorithmic one—$f#_{\text{best}}$ might not be finitely computable!

- **Parity example continued:**
  - $\text{succ#}_{\text{best}}(\text{even}) = \alpha(\text{succ}(\gamma(\text{even}))) = \alpha(\text{succ}\{2n \mid n\geq0\}) = \alpha\{2n+1 \mid n\geq0\} = \text{odd}$
  - Question: what about other operators on Nat, e.g., *, / ?
Completeness of Approximation (skip)

Given $C^{\alpha, \gamma}A$, and function $f : C \rightarrow C$,

- Function $f^\#: A \rightarrow A$ is sound with respect to $f$ iff
  - For all $c \in C$, $\alpha(f(c)) \leq f^\#(\alpha(c))$
  - For all $a \in A$, $f(\gamma(a)) \leq \gamma(f^\#(a))$

- Function $f^\#: A \rightarrow A$ is forwards($\gamma$) complete with respect to $f$ iff
  - For all $a \in A$, $f(\gamma(a)) = \gamma(f^\#(a))$
  - That is, $\gamma(A)$ is closed under $f : f(\gamma(A)) \subseteq \gamma(A)$

- Function $f^\#: A \rightarrow A$ is backwards($\alpha$) complete with respect to $f$ iff
  - For all $c \in C$, $\alpha(f(c)) = f^\#(\alpha(c))$
  - That is, $\alpha$ partitions $C$ into equivalence classes: $\alpha(c) = \alpha(c')$ implies $\alpha(f(c)) = \alpha(f(c'))$

- For an $f^\#$ to be (forwards or backwards) complete, it must equal $f^\#_{\text{best}} = \alpha(f(\gamma(a)))$
  - The structure of $C^{\alpha, \gamma}A$ and $f : C \rightarrow C$ determines whether $f^\#$ is complete.
Transfer Functions and Computation steps

- Each program transition from program point $p_i$ to $p_j$ has an associated *transfer function*, $f_{ij}: C \rightarrow C$ (or $f_{#ij}: A \rightarrow A$), which describes the associated computation.
  - This defines a computation step of the form, $(p_i, s) \rightarrow (p_j, f_{ij}(s))$

- **Example:**
  - Assignment $p_0: x = x + 1; p_1: \cdots$ has the transfer function
    $$f_{01}(<\ldots x: n\ldots>) = <\ldots x: n+1\ldots>$$
  - For multiple transitions in conditionals, attach a transfer function to each possible transition (branch) to “filter” the data that arrives at a program point.
    e.g. $p_0$: cases $x \leq y$: $p_1: y = y - x$;
      $y \leq x$: $p_2: x = x - y$; end
    - $f_{p1}(s) = \text{if } s[x] \leq s[y] \text{ then } s \text{ else bot};$ (filter out $s$ unless $s[x] \leq s[y]$)
    - $f_{p2}(s) = \text{if } s[y] \leq s[x] \text{ then } s \text{ else bot};$ (filter out $s$ unless $s[y] \leq s[x]$)
Execution Traces

- An *execution trace* is a (possibly infinite) sequence, 
  \[(p_0,s_0)\rightarrow(p_1,s_1)\rightarrow\cdots\rightarrow(p_j,s_j)\rightarrow\cdots\] 
  s.t.
  - for all \(i \geq 0\): \((p_i,s_i) \rightarrow p_{\text{succ}(i)}, f_{\text{succ}(i)}(s_i)\)
  - No \(s_i\) equals bot

Two concrete traces
\((p_i,v)\) means \((p_i,x=v)\):

P0: while (x != 1) {
P1: if Even(x)
P2: x = x div2;
P3: else
   x = 3*x + 1;
P5: exit;
Using Approximation to build abstract traces

Abstract over approximating trace:

1. Each concrete transition is generated by an $f_{ij}$;
2. Each abstract transition is generated by the corresponding $f\#ij$.

- Each concrete transition, $(pi,s) \rightarrow (pj,fij(s))$, is reproduced by a corresponding abstract transition, $(pi,a) \rightarrow (pj,f\#ij(a))$, where $s \in \gamma(a)$
- The traces embedded in the abstract trace tree “cover” (simulate) the concrete traces
Shape Analysis

- **Goal**
  - To obtain a finite representation of the memory storage

- The analysis result can be used for
  - Detection of pointer aliasing
  - Detection of sharing between structures
  - Software development tools
    - Detection of pointer errors, e.g. dereferences of nil-pointers
  - Program verification
    - E.g., reverse transforms a non-cyclic list to a non-cyclic list
The Concrete Solution Space

- Model the memory (stack and heap)
  - Storage of local variables
    \( \text{Stack} = \text{Var} \rightarrow (\text{Value} \cup \text{Loc}) \)
    Map each local variable into a value or a unique location
  - The heap storage
    \( \text{Heap} = (\text{Loc} \times \text{Sel}) \rightarrow (\text{Value} \cup \text{Loc}) \)
    Map pairs of locations and selectors to values or locations

- Model the operational semantics of programs
  - Program state: \( \text{State} = \text{ProgramPoint} \times \text{Stack} \times \text{Heap} \)
    **Example:** \((p1, (x:3,y:Ly), ((Ly,val):5))\) is a program state
  - Each statement modifies Stack and Heap of the previous state
    - Stmt: \( \text{State} \rightarrow \text{State} \)
Building Abstract Domains

- Given an unordered set, D, of concrete data values, we might ask,
  - “What are the properties about D that I wish to calculate?
  - Can I relate these properties $a \in A$, to elements $d \in D$ via a UG-closed binary relation, $R: D \times A$?

- Given a set, A, and a binary relation, $R: D \times A$
  - Define $\gamma: A \rightarrow \text{Power}(D)$ as $\gamma(a) = \{d \in D \mid d \, R \, a\}$
  - Define partial ordering on $A$: $a \leq a'$ iff $\gamma(a) \leq \gamma(a')$
    - If there are distinct $a$ and $a'$ such that $\gamma(a) = \gamma(a')$, then merge them to force U-closure
  - Ensure that $\gamma(A)$ is a Moore family by adding greatest-lower-bound elements to A as needed.
    - This forces G-closure
  - Use the existing machinery to define the Galois connection between Power(D) and A
Abstracting the Program State

- Build a binary relation, $R_d: \text{Data} \times \text{AbsData}$
  - $R_v: \text{Value} \rightarrow \text{AbsValue}$; $R_l: \text{Loc} \rightarrow \text{AbsLoc}$
  - May ignore the values of non-pointer variables.

- Build induced Galois connection, $\text{Power(Data)} <\alpha, \gamma> \text{AbsData}$, we can
  - Build Galois connections that abstract the concrete data
    $<x_i : v_i> R_s <x_i : a_i> \text{ iff } v_i R_d a_i$
    **Example**: $<x:3, y:4> R_s <x:any, y:any>$
  - A program point is abstracted to itself: $p R_p p$,
    the abstract domain of program points is $\text{ProgramPoint} \cup \{\text{top, bot}\}$ (to make it a complete lattice)
  - Finally, we can relate each concrete state to an abstract one:
    $(p, s) R_s (p', s') \text{ iff } p = p' \text{ and } s R_s s'$
Shape Graphs

- Shape analysis uses a shape graph to abstract the memory storage
  - Graph nodes denote a finite number of abstract locations:
    - $\text{Alloc} = \{\text{Nx} \mid \text{Nx is pointed to by a set of local variables}\} \cup N_\phi$
      - $\text{Nx}$: the node represents all concrete Locations referred to by variables in $x$
      - $N_\phi$: abstract summary location (all the other locations)
    - Each graph node abstracts a distinctive set of concrete Locations
      - If variables $x$ and $y$ may be aliased, they must share a single graph node
  - A graph edge $\text{sel}$ connect nodes $n_1$ and $n_2$ if $n_2$ is pointed to by $n_1.\text{sel}$

Diagram:

- $x$ connects $N\{x\}$ to $N\{z\}$
- $y$ connects $N\{y\}$ to $N_\phi$
- $Z$ connects $N\{z\}$ to $N_\phi$
- $\text{next}$ labels the edges
Abstraction of Program States

- Abstraction of memory storage
  - Abstract Stack
    \[ \text{AbsStack} = \text{Var} \rightarrow \text{ALoc} \]
    Map each pointer variable into a unique abstract location (a shape graph node)
  - Abstract heap
    \[ \text{AbsHeap} = (\text{ALoc} \times \text{Sel}) \rightarrow (\text{ALoc}) \]
    Mapping pairs of abs locations and selectors to abs locations
  - Sharing information
    \[ \text{IS} : \text{ALoc} \rightarrow \{ \text{yes, no} \} \]
    For each abstract location in the shape graph, is it shared by pointers in the heap?
    - If \( \text{IS}(N_x) = \text{yes} \), then \( N_x \) must have an incoming edge from \( N_\phi \) or have more than one incoming edges

- Transfer functions: \( P(\text{AbsState}) \rightarrow P(\text{AbsState}) \)
  - Program state: \( \text{AbsState} = \text{ProgramPoint} \times \text{AbsStack} \times \text{AbsHeap} \times \text{IS} \)
  - Each statement modifies mappings in the previous state
Transfer functions(1)

- $x = \text{nil}$
  - $F(S, H, IS) = (S', H', IS')$ where $(S', H', IS')$ is obtained from $(S, H, IS)$ by:
    - Removing $x$ from all mappings (killing all previous info. about $x$)
    - Merging all $N\phi$ nodes
Transfer functions(2)

- $x = y$
  - $F(S, H, IS) = (S', H', IS')$ where
    - $(S', H', IS')$ is obtained by modifying mappings for $x$ to be identical to those for $y$
Transfer functions (3)

- \( x = y \texttt{.sel} \)
  - Remove the old binding for \( x \)
  - Establish a new binding for \( x \) to be the same as \( y \texttt{.sel} \)
    - If there is no abstract location defined for \( y \)
      - Error: dereference a null pointer
    - If there is an abstract location \( \texttt{N} y \) s.t. \( S[y] = \texttt{N} y \), but there is no abstract location for \( (\texttt{N} y, \texttt{sel}) \)
      - Error dereference a non-existing field
    - If there exist abstract locations \( \texttt{N} y \) and \( \texttt{N} z \) s.t. \( S[y] = \texttt{N} y \) and \( H[\texttt{N} y, \texttt{sel}] = \texttt{N} z \).
      - Modify the mappings so that \( x \) points to \( \texttt{N} z \)
      - If \( \texttt{N} z = \texttt{N} \phi \), create a new node \( \texttt{N} \{x\} \) for \( x \) --- may need to create multiple shape graphs to cover different cases

- Other transfer functions
  - E.g. \( x \texttt{.sel} = y \); \( x \texttt{.sel} = \texttt{nil} \); \( \texttt{allocate}(x) \);