Last time... Dataflow Analysis

- Information flows along (potential) execution paths
- Conservative approximation of all possible program executions
- Can be viewed as a sequence of transformations on program state
  - Every statement (block) is associated with two abstract states: input state, output state
  - Input/output state represents all possible states that can occur at the program point
  - Representation is finite
  - Different problems typically use different representations

Control-Flow Graph

1: y := x;
2: z := 1;
3: while y > 0 {
  4: z := z * y;
  5: y := y - 1
}
6: y := 0

Executions

1: y := x;
2: z := 1;
3: while y > 0 {
  4: z := z * y;
  5: y := y - 1
}
6: y := 0
Input/output Sets

1: y := x;
2: z := 1;
3: while y > 0 {
4:    z := z * y;
5:    y := y − 1
6: y := 0
}

Transfer Functions

1: y := x
2: z := 1
3: y > 0
4: z = z * y
5: y = y − 1
6: y := 0

Kill/Gen formulation for Reaching Definitions

<table>
<thead>
<tr>
<th>Block</th>
<th>out (lab)</th>
</tr>
</thead>
<tbody>
<tr>
<td>{x := a}^m</td>
<td>m(lab) \cup { (x, l) \mid l \in \text{Lab} } \cup { (x, lab) }</td>
</tr>
<tr>
<td>{skip}^m</td>
<td>m(lab)</td>
</tr>
<tr>
<td>b^m</td>
<td>m(lab)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Block</th>
<th>kill</th>
<th>gen</th>
</tr>
</thead>
<tbody>
<tr>
<td>{x := a}^m</td>
<td>{ (x, l) \mid l \in \text{Lab} } \cup { (x, lab) }</td>
<td></td>
</tr>
<tr>
<td>{skip}^m</td>
<td>\emptyset</td>
<td>\emptyset</td>
</tr>
<tr>
<td>b^m</td>
<td>\emptyset</td>
<td>\emptyset</td>
</tr>
</tbody>
</table>

Solving Gen/Kill Equations

\[ \text{OUT}[\text{ENTRY}] = \emptyset; \]

for each basic block B other than ENTRY \[ \text{OUT}[B] = \emptyset; \]

while (changes to any OUT occur) \{ \]

for each basic block B other than ENTRY \{ \]

\[ \text{OUT}[B] = (\text{IN}[B] \setminus \text{killB}) \cup \text{genB} \]

\[ \text{IN}[B] = \bigcup_{p \in \text{pred}(B)} \text{OUT}[p] \]

\}\}

- Designated block Entry with OUT[Entry] = \emptyset
- \text{pred}(B) = \text{predecessor nodes of B in the control flow graph}
Available Expressions Analysis

- Property space
  - in_{AE}, out_{AE}: Lab \rightarrow \mathcal{P}(\text{AExp})
  - Mapping a label to set of arithmetic expressions available at that label

- Dataflow equations
  - Flow equations – how to join incoming dataflow facts
  - Effect equations – given an input set of expressions S, what is the effect of a statement

For each program point, which expressions must have already been computed, and not later modified, on all paths to the program point

Some required notation

- blocks: Stmt \rightarrow P(Blocks)
- blocks([x := a]_\text{lab}) = [[x := a]_\text{lab}]
- blocks([skip]_\text{lab}) = [[skip]_\text{lab}]
- blocks([S_1; S_2]) = blocks(S_1) \cup blocks(S_2)
- blocks(if [b]_\text{lab} then S_1 else S_2) = [[b]_\text{lab}] \cup blocks(S_1) \cup blocks(S_2)
- blocks(while [b]_\text{lab} do S) = [[b]_\text{lab}] \cup blocks(S)

FV: (\text{BExp} \cup \text{AExp}) \rightarrow \text{Var}
Variables used in an expression

\text{AExp}(a) = \text{all non-unit expressions in the arithmetic expression a}
similarly \text{AExp}(b) for a boolean expression b

Available Expressions Analysis

- in_{AE}(\text{lab}) =
  - \emptyset when \text{lab} is the initial label
  - \cap \{ out_{AE}(\text{lab}') | \text{lab}' \in \text{pred}(\text{lab}) \} otherwise
- out_{AE}(\text{lab}) = ...

<table>
<thead>
<tr>
<th>Block</th>
<th>out (lab)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[x := a]_\text{lab}</td>
<td>in(\text{lab}) \cup { x' \in \text{AExp}</td>
</tr>
<tr>
<td>[skip]_\text{lab}</td>
<td>in(\text{lab})</td>
</tr>
<tr>
<td>(b) |_\text{lab}</td>
<td>in(\text{lab}) \cup \text{AExp}(b)</td>
</tr>
</tbody>
</table>

From now on going to drop the AE subscript when clear from context
Transfer Functions

1. \(x = a + b\)
   \(\text{out}(1) = \text{in}(1) \cup \{a + b\}\)
2. \(y := a \cdot b\)
   \(\text{out}(2) = \text{in}(2) \cup \{a \cdot b\}\)
3. \(y > a + b\)
4. \(a = a + 1\)
5. \(x = a + b\)

\[\text{out}(1) = \text{in}(1) \cup \{a + b\}\]
\[\text{out}(2) = \text{in}(2) \cup \{a \cdot b\}\]
\[\text{out}(4) = \text{in}(4) \cup \{a + b, a \cdot b, a + 1\}\]
\[\text{out}(5) = \text{in}(5) \cup \{a + b\}\]

Solution

1. \(x = a + b\)
2. \(y := a \cdot b\)
3. \(y > a + b\)
4. \(a = a + 1\)
5. \(x = a + b\)

\[\text{in}(2) = \text{out}(1) = \{a + b\}\]
\[\text{out}(2) = \{a + b, a \cdot b\}\]
\[\text{out}(4) = \{a + b\}\]
\[\text{out}(5) = \{a + b\}\]

Kill/Gen

<table>
<thead>
<tr>
<th>Block</th>
<th>out (lab)</th>
</tr>
</thead>
<tbody>
<tr>
<td>([x := a])</td>
<td>(\text{in}(lab) \setminus {a' \in AExp \mid x \in FV(a')} \cup {a' \in AExp(a) \mid x \notin FV(a')})</td>
</tr>
<tr>
<td>[skip]</td>
<td>(\text{in}(lab))</td>
</tr>
<tr>
<td>[b]</td>
<td>(\text{in}(lab) \cup \text{AExp}(b))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Block</th>
<th>kill gen</th>
</tr>
</thead>
<tbody>
<tr>
<td>([x := a])</td>
<td>({a' \in AExp \mid x \in FV(a')}) ({a' \in AExp(a) \mid x \notin FV(a')})</td>
</tr>
<tr>
<td>[skip]</td>
<td>(\emptyset) (\emptyset)</td>
</tr>
<tr>
<td>[b]</td>
<td>(\emptyset) (\emptyset)</td>
</tr>
</tbody>
</table>

\(\text{out}(lab) = \text{in}(lab) \setminus \text{kill}(B^\text{lab}) \cup \text{gen}(B^\text{lab})\)

\(B^\text{lab} = \text{block at label } \text{lab}\)

Reaching Definitions Revisited

<table>
<thead>
<tr>
<th>Block</th>
<th>out (lab)</th>
</tr>
</thead>
<tbody>
<tr>
<td>([x := a])</td>
<td>(\text{in}(lab) \setminus {(x,l) \mid l \in \text{Lab}} \cup {(x,lab)})</td>
</tr>
<tr>
<td>[skip]</td>
<td>(\text{in}(lab))</td>
</tr>
<tr>
<td>[b]</td>
<td>(\text{in}(lab))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Block</th>
<th>kill gen</th>
</tr>
</thead>
<tbody>
<tr>
<td>([x := a])</td>
<td>({(x,l) \mid l \in \text{Lab}}) ({(x,lab)})</td>
</tr>
<tr>
<td>[skip]</td>
<td>(\emptyset) (\emptyset)</td>
</tr>
<tr>
<td>[b]</td>
<td>(\emptyset) (\emptyset)</td>
</tr>
</tbody>
</table>

For each program point, which assignments may have been made and not overwritten, when program execution reaches this point along some path.
Why solution with smallest sets?

After simplification: \( \text{in}(z) = \text{in}(z) \cup \{(x,?), (y,?), (z,1)\} \)

Many solutions: any superset of \( \{(x,?), (y,?), (z,1)\} \)

Live Variables

For each program point, which variables may be live at the exit from the point.

Live Variables

For each program point, which variables may be live at the exit from the point.
Live Variables: solution

\[
\begin{align*}
[x := 2]_1; \\
[y := 4]_2; \\
[x := 3]_3; \\
\text{if } [y > x]^4 \text{ then } [z := y]^5; \\
\text{else } [z := y*y]^6; \\
[x := 2]^7
\end{align*}
\]

\begin{center}
\begin{tabular}{|c|c|c|}
\hline
Block & kill & gen \\
\hline
\{x := a\} & \{x\} & \{FV(a)\} \\
\{skip\} & \emptyset & \emptyset \\
\{b\} & \emptyset & \{FV(b)\} \\
\hline
\end{tabular}
\end{center}

Why solution with smallest set?

\begin{center}
\begin{tikzpicture}
\node (1) at (0,0) {in(1) \textcolor{black}{=} \emptyset}; \\
\node (2) at (1,0) {1: \textcolor{red}{x \leftarrow 2}}; \\
\node (3) at (2,0) {2: \textcolor{red}{y \leftarrow 4}}; \\
\node (4) at (3,0) {3: \textcolor{red}{x \leftarrow 1}}; \\
\node (5) at (4,0) {4: \textcolor{red}{y > x}}; \\
\node (6) at (5,0) {5: \textcolor{red}{z \leftarrow y}}; \\
\node (7) at (6,0) {6: \textcolor{red}{z \leftarrow y*y}}; \\
\node (8) at (7,0) {7: \textcolor{red}{x \leftarrow z}}; \\
\node (9) at (0,-1) {out(1) \textcolor{black}{=} in(2) \cup \emptyset}; \\
\node (10) at (1,-1) {out(2) \textcolor{black}{=} in(3) + \{y\}}; \\
\node (11) at (2,-1) {out(3) \textcolor{black}{=} in(4) + \{x, y\}}; \\
\node (12) at (3,-1) {out(4) \textcolor{black}{=} \{x, y\}}; \\
\node (13) at (4,-1) {out(5) \textcolor{black}{=} \{z\}}; \\
\node (14) at (5,-1) {out(6) \textcolor{black}{=} \{z\}}; \\
\node (15) at (6,-1) {out(7) \textcolor{black}{=} \emptyset}; \\
\node (16) at (0,-0.5) {in(1) \textcolor{red}{=} \{x\}}; \\
\node (17) at (1,-0.5) {in(2) \textcolor{red}{=} \{y\}}; \\
\node (18) at (2,-0.5) {in(3) \textcolor{red}{=} \{x\}}; \\
\node (19) at (3,-0.5) {in(4) \textcolor{red}{=} \{x, y\}}; \\
\node (20) at (4,-0.5) {in(5) \textcolor{red}{=} \{z\}}; \\
\node (21) at (5,-0.5) {in(6) \textcolor{red}{=} \{z\}}; \\
\node (22) at (6,-0.5) {in(7) \textcolor{red}{=} \emptyset}; \\
\end{tikzpicture}
\end{center}

After simplification: \(\text{in}(x) = \text{in}(1) \cup \{x\}\)

Many solutions: any superset of \(\{x\}\)

Monotone Frameworks

\begin{itemize}
  \item \(\bigcup\) is \(\cup\) or \(\cap\)
  \item CFG edges go either forward or backwards
  \item Entry labels are either initial program labels or final program labels (when going backwards)
  \item Initial is an initial state (or final when going backwards)
  \item \(f_{lab}\) is the transfer function associated with the blocks \(B^{lab}\)
\end{itemize}

Forward vs. Backward Analyses

\begin{center}
\begin{tikzpicture}
\node (1) at (0,0) {1: \textcolor{red}{x \leftarrow 2}}; \\
\node (2) at (1,0) {2: \textcolor{red}{y \leftarrow 4}}; \\
\node (3) at (2,0) {3: \textcolor{red}{x \leftarrow x + 1}}; \\
\node (4) at (3,0) {4: \textcolor{red}{y > x}}; \\
\node (5) at (4,0) {5: \textcolor{red}{z \leftarrow y}}; \\
\node (6) at (5,0) {6: \textcolor{red}{z \leftarrow y*y}}; \\
\node (7) at (6,0) {7: \textcolor{red}{x \leftarrow z}}; \\
\node (8) at (0,-0.5) {in(1) \textcolor{black}{=} \{x, y\}}; \\
\node (9) at (1,-0.5) {in(2) \textcolor{black}{=} \{y\}}; \\
\node (10) at (2,-0.5) {in(3) \textcolor{black}{=} \{x\}}; \\
\node (11) at (3,-0.5) {in(4) \textcolor{black}{=} \{x, y\}}; \\
\node (12) at (4,-0.5) {in(5) \textcolor{black}{=} \{z\}}; \\
\node (13) at (5,-0.5) {in(6) \textcolor{black}{=} \{z\}}; \\
\node (14) at (6,-0.5) {in(7) \textcolor{black}{=} \emptyset}; \\
\node (15) at (0,-1) {out(1) \textcolor{black}{=} \{x\}}; \\
\node (16) at (1,-1) {out(2) \textcolor{black}{=} \{y\}}; \\
\node (17) at (2,-1) {out(3) \textcolor{black}{=} \{x\}}; \\
\node (18) at (3,-1) {out(4) \textcolor{black}{=} \{x, y\}}; \\
\node (19) at (4,-1) {out(5) \textcolor{black}{=} \{z\}}; \\
\node (20) at (5,-1) {out(6) \textcolor{black}{=} \{z\}}; \\
\node (21) at (6,-1) {out(7) \textcolor{black}{=} \emptyset}; \\
\end{tikzpicture}
\end{center}
Must vs. May Analyses

- When $\sqcap$ is $\cap$ - must analysis
  - Want largest sets that solves the equation system
  - Properties hold on all paths reaching a label (exiting a label, for backwards)

- When $\sqcap$ is $\cup$ - may analysis
  - Want smallest sets that solve the equation system
  - Properties hold at least on one path reaching a label (existing a label, for backwards)

Example: Reaching Definition

- $L = \varphi(Var \times Lab)$ is partially ordered by $\subseteq$
- $\sqcap$ is $\sqcap$
- $L$ satisfies the Ascending Chain Condition because $Var \times Lab$ is finite (for a given program)

Example: Available Expressions

- $L = \varphi(AExp)$ is partially ordered by $\subseteq$
- $\sqcap$ is $\cap$
- $L$ satisfies the Ascending Chain Condition because $AExp$ is finite (for a given program)

Analyses Summary

<table>
<thead>
<tr>
<th>Reaching Definitions</th>
<th>Available Expressions</th>
<th>Live Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L$</td>
<td>$\varphi(Var \times Lab)$</td>
<td>$\varphi(AExp)$</td>
</tr>
<tr>
<td>$\sqcap$</td>
<td>$\sqcap$</td>
<td>$\cup$</td>
</tr>
<tr>
<td>$\sqcup$</td>
<td>$\cup$</td>
<td>$\sqcap$</td>
</tr>
<tr>
<td>$\bot$</td>
<td>$\bot$</td>
<td>$AExp$</td>
</tr>
<tr>
<td>Initial</td>
<td>${ (x,?) \mid x \in Var }$</td>
<td>$\varnothing$</td>
</tr>
<tr>
<td>Entry labels</td>
<td>${ init }$</td>
<td>${ init }$</td>
</tr>
<tr>
<td>Direction</td>
<td>Forward</td>
<td>Forward</td>
</tr>
<tr>
<td>F</td>
<td>$f : L \rightarrow L \sqcup g : (val) = (val \setminus k) \sqcup g$</td>
<td></td>
</tr>
<tr>
<td>$f_{lab}$</td>
<td>$f_{lab}(val) = (val \setminus kill) \sqcup gen$</td>
<td></td>
</tr>
</tbody>
</table>
Analyses as Monotone Frameworks

- Property space
  - Powerset
  - Clearly a complete lattice

- Transformers
  - Kill/gen form
  - Monotone functions (let's show it)

Monotonicity of Kill/Gen transformers

- Have to show that \( x \sqsubseteq x' \) implies \( f(x) \sqsubseteq f(x') \)
- Assume \( x \sqsubseteq x' \), then for kill set \( k \) and gen set \( g \)
  \[ (x \setminus k) \cup g \sqsubseteq (x' \setminus k) \cup g \]
- Technically, since we want to show it for all functions in \( F \), we also have to show that the set is closed under function composition

Distributivity of Kill/Gen transformers

- Have to show that \( f(x \sqcup y) \sqsubseteq f(x) \sqcup f(y) \)
- \( f(x \sqcup y) = ((x \sqcup y) \setminus k) \cup g \)
  \[ = ((x \setminus k) \sqcup (y \setminus k)) \cup g \]
  \[ = (((x \setminus k) \cup g) \sqcup ((y \setminus k) \cup g)) \]
  \[ = f(x) \sqcup f(y) \]
- Used distributivity of \( \sqcup \) and \( \sqcup \)
  - Works regardless of whether \( \sqcup \) is \( \sqcup \) or \( \sqcap \)

Points-to Analysis

- Many flavors
- PWHILE language

\[ p \in \text{PExp} \quad \text{pointer expressions} \]
\[ a ::= x \mid n \mid a_1 \text{ op } a_2 \mid \& x \mid * x \mid \text{nil} \]
\[ S ::= [x := a]^{\text{**}} \]
\[ |(\text{skip})^{\text{**}} | S_1;S_2 \]
\[ | \text{if } [b] \text{ then } S_1 \text{ else } S_2 \]
\[ | \text{while } [b]^{\text{**}} \text{ do } S_1 \]
\[ | x = \text{malloc} \]
Points-to Analysis

- **Aliases**
  - Two pointers \( p \) and \( q \) are aliases if they point to the same memory location.

- **Points-to pair**
  - \( (p,q) \) means \( p \) holds the address of \( q \).

- **Points-to pairs and aliases**
  - \( (p,q) \) and \( (r,q) \) means that \( p \) and \( r \) are aliases.

- **Challenge**: no a priori bound on the set of heap locations.

---

Terminology Example

\[
\begin{align*}
[x := &z]^3 \quad [y := &z]^2 \\
[w := &y]^3 \quad [r := w]^6
\end{align*}
\]

Points-to pairs: \((x,z), (y,z), (w,y), (r,y)\)

Aliases: \((x,y), (r,w)\)

---

(May) Points-to Analysis

- **Property Space**
  - \( L = \{ \varphi(\text{Var x Var}), \subseteq, \cup, \cap, \emptyset, \text{Var x Var} \} \)

- **Transfer functions**

<table>
<thead>
<tr>
<th>Statement</th>
<th>out(lab)</th>
</tr>
</thead>
<tbody>
<tr>
<td>([p = &amp;x]^3)</td>
<td>in(lab) U {(p,x)}</td>
</tr>
<tr>
<td>([p = q]^3)</td>
<td>in(lab) U {(p,x) \</td>
</tr>
<tr>
<td>([*p = q]^3)</td>
<td>in(lab) U {(r,x) \</td>
</tr>
<tr>
<td>([p = *q]^3)</td>
<td>in(lab) U {(p,r) \</td>
</tr>
</tbody>
</table>

---

(May) Points-to Analysis

- **What to do with \texttt{malloc}**?

- **Need some static naming scheme for dynamically allocated objects**

- **Single name for the entire heap**
  - \([([p = \texttt{malloc}]^3)(S) = S \cup \{(p,H)\} \]

- **Name based on static allocation site**
  - \([([p = \texttt{malloc}]^3)(S) = S \cup \{(p,lab)\} \]
(May) Points-to Analysis

# Allocation Sites

- Divide the heap into a fixed partition based on allocation site
- All objects allocated at the same program point represented by a single "abstract object"

# Weak Updates

```plaintext
Statement | out(lab) |
-----------|----------|
[p = &x]   | in(lab) U \{(p,x)\} |
[p = q]    | in(lab) U \{(p,x) \in \text{in(lab)} \} |
[p^* = q]  | \\{x\} \in \text{in(lab)} and (p,r) \in \text{in(lab)} \} |
[p^* = *q] | \\{x\} \in \text{in(lab)} and (x,r) \in \text{in(lab)} \} |
```

```plaintext
[x := malloc]; // A1
[y := malloc]; // A2
(if [x == y] then
  [z := x];
else
  [z := y];
);
```

Single name H for the entire heap

Allocation-site based naming (using A_{lab} instead of just "lab" for clarity)

```plaintext
[x := malloc]; // A1
[y := malloc]; // A2
(if [x == y] then
  [z := x];
else
  [z := y];
);
```
(May) Points-to Analysis

- Fixed partition of the (unbounded) heap to static names
  - Allocation sites
  - Types
  - Calling contexts
  - ...
- What we saw so far – flow-insensitive
  - Ignoring the structure of the flow in the program

Flow-sensitive vs. Flow-insensitive Analyses

- Flow sensitive: respect program flow
  - a separate set of points to pairs for every program point
  - the set at a point represents possible may-aliases on some path from entry to the program point
- Flow insensitive: assume all execution orders are possible, abstract away order between statements

So far...

- Intra-procedural analysis
- How are we going to deal with procedures?
- Inter-procedural analysis

Interprocedural Analysis

- The effect of calling a procedure is the effect of executing its body
Solution Attempt #1

- Inline callees into callers
  - End up with one big procedure
  - CFGs of individual procedures = duplicated many times
  - Good: it is precise
    - distinguishes different calls to the same function
  - Bad
    - exponential blow-up, not efficient
    - doesn't work with recursion
Solution Attempt #2

- Build a "supergraph" = inter-procedural CFG
- Replace each call from P to Q with:
  - An edge from point before the call (call point) to Q's entry point
  - An edge from Q's exit point to the point after the call (return pt)
  - Add assignments of actuals to formals, and assignment of return value
- Good: efficient
  - Graph of each function included exactly once in the supergraph
  - Works for recursive functions (although local variables need additional treatment)
- Bad: imprecise, "context-insensitive"
  - The "unrealizable paths problem": dataflow facts can propagate along infeasible control paths

Unrealizable Paths

Interprocedural Analysis

```plaintext
begin
proc p() is
  x := a + 1
end

a := 7

[call p()]

print x

[call p()]

print a
end
```

IVP: Interprocedural Valid Paths

- IVP: all paths with matching calls and returns
- And prefixes
Valid Paths

\begin{align*}
\text{foo}() & \rightarrow \text{bar}() \\
\text{bar}() & \rightarrow \text{zoo}() \\
\text{bar}() & \rightarrow \text{200}() \\
\end{align*}

Interprocedural Valid Paths

- IVP set of paths
  - Start at program entry
  - Only considers matching calls and returns
  - aka, valid

- Can be defined via context free grammar
  - matched ::= matched (matched) | \epsilon
  - valid ::= valid (matched | matched
  - paths can be defined by a regular expression

The Join-Over-Valid-Paths (JVP)

- vpaths(n) all valid paths from program start to n
- JVP[n] = \{\epsilon_{e_1, e_2, \ldots, e_n} \ (initial) \mid (e_{e_1, e_2, \ldots, e_n} \in vpaths(n))\}
- JVP \subseteq JFP
  - In some cases the JVP can be computed
  - (Distributive problem)

Sharir and Pnueli ‘82

- Call String approach
  - Blend interprocedural flow with intra procedural flow
  - Tag every dataflow fact with call history

- Functional approach
  - Determine the effect of a procedure
    - E.g., in/out map
  - Treat procedure invocations as “super ops”
The Call String Approach

- Record at every node a pair \((l, c)\) where \(l \in L\) is the dataflow information and \(c\) is a suffix of unmatched calls
- Use Chaotic iterations
- To guarantee termination limit the size of \(c\) (typically 1 or 2)
- Emulates inline (but no code growth)
- Exponential in size of \(c\)

```
begin
  proc p() is
    if \([b]\) then (
      \([a := a - 1]\)
      \([\text{call } p()]\)
    )
    \([x := -2 \times a + 5]\)
  end

  \([x := a + 1]\)
  \([\text{call } p()]\)
  \([\text{print } x]\)
  \([a := a + 1]\)
  \([\text{call } p()]\)
  \([\text{print } x]\)
end
```

The Functional Approach

- The meaning of a procedure is mapping from states into states
- The abstract meaning of a procedure is function from an abstract state to abstract states
begin
proc p() is¹
  if [b]² then (\n    [a := a - 1]³ [x := 0, a := 0]
    (call p())⁴  
    [a := a + 1]⁵ [x := -2 * a + 5]⁶ 
    )
  [x := -2 * a + 5]⁷ 
end⁸
[a=7]; [call p()]⁹; [print(x)]¹⁰
end

begin
proc p() is¹
  if [b]² then (\n    [a := a - 1]³ [x := 0, a := 0]
    (call p())⁴  
    [a := a + 1]⁵ [x := -2 * a + 5]⁶ 
    )
  [x := -2 * a + 5]⁷ 
end⁸
[a=7]; [call p()]⁹; [print(x)]¹⁰
end

Functional Approach: Main Idea

- Iterate on the abstract domain of functions from L to L
- Two phase algorithm
  - Compute the dataflow solution at the exit of a procedure as a function of the initial values at the procedure entry (functional values)
  - Compute the dataflow values at every point using the functional values
- Computes JVP for distributive problems