Lecture 16 – Dataflow Analysis

THEORY OF COMPILATION

Eran Yahav


Reference: Dragon 9, 12
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: \( z := z \times y; \)
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: \( y := x \)

\[ \text{out}(1) = \text{in}(1) \setminus \{(y,l) \mid l \in \text{Lab}\} \cup \{(y,1)\} \]

2: \( z := 1 \)

\[ \text{out}(2) = \text{in}(2) \setminus \{(z,l) \mid l \in \text{Lab}\} \cup \{(z,2)\} \]

3: \( y > 0 \)

\[ \text{out}(3) = \text{in}(3) \]

4: \( z = z \cdot y \)

\[ \text{out}(4) = \text{in}(4) \setminus \{(z,l) \mid l \in \text{Lab}\} \cup \{(z,4)\} \]

5: \( y = y - 1 \)

\[ \text{out}(5) = \text{in}(5) \setminus \{(y,l) \mid l \in \text{Lab}\} \cup \{(y,5)\} \]

6: \( y := 0 \)

\[ \text{out}(6) = \text{in}(6) \setminus \{(y,l) \mid l \in \text{Lab}\} \cup \{(y,6)\} \]

\[ \text{in}(1) = \{(x,\_), (y,\_), (z,\_)\} \]
\[ \text{in}(2) = \text{out}(1) \]
\[ \text{in}(3) = \text{out}(2) \cup \text{out}(5) \]
\[ \text{in}(4) = \text{out}(3) \]
\[ \text{in}(5) = \text{out}(4) \]
\[ \text{in}(6) = \text{out}(3) \]
Kill/Gen formulation for Reaching Definitions

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

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

\[ \text{OUT}[\text{ENTRY}] = \emptyset; \]
\[ \text{for (each basic block B other than ENTRY) OUT}[B] = \emptyset; \]
\[ \text{while (changes to any OUT occur) \{} \]
\[ \text{for (each basic block B other than ENTRY) \{} \]
\[ \text{OUT}[B] = (\text{IN}[B] \setminus \text{killB}) \cup \text{genB} \]
\[ \text{IN}[B] = \bigcup_{p \in \text{pred}(B)} \text{OUT}[p] \]
\[ \text{\}} \]
\[ \text{\}} \]

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

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

\[
\begin{align*}
\text{x := a+b} & \quad ^{1} ; \\
\text{y := a*b} & \quad ^{2} ; \\
\text{while y > a+b} & \quad ^{3} ( \\
\quad \text{a := a + 1} & \quad ^{4} ; \\
\quad \text{x := a + b} & \quad ^{5} \\
\) \\
\end{align*}
\]

(a+b) always available at label 3

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 → P(Blocks)
blocks([x := a]_{lab}) = {[x := a]_{lab}}
blocks([skip]_{lab}) = {[skip]_{lab}}
blocks(S_1; S_2) = blocks(S_1) ∪ blocks(S_2)
blocks(if [b]_{lab} then S_1 else S_2) = {[b]_{lab}} ∪ blocks(S_1) ∪ blocks(S_2)
blocks(while [b]_{lab} do S) = {[b]_{lab}} ∪ blocks(S)

FV: (BExp ∪ AExp) → Var
Variables used in an expression

AExp(a) = all non-unit expressions in the arithmetic expression a
similarly AExp(b) for a boolean expression b
Available Expressions Analysis

- **Property space**
  - \( \text{in}_{AE}, \text{out}_{AE}: \text{Lab} \rightarrow \mathcal{P}(\text{AExp}) \)
  - 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
Available Expressions Analysis

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

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

out(1) = in(1) \cup \{ a+b \}$

2: $y := a*b$

out(2) = in(2) \cup \{ a*b \}$

3: $y > a+b$

out(3) = in(3) \cup \{ a+b \}$

4: $a = a+1$

out(4) = in(4) \setminus \{ a+b, a*b, a+1 \}$

5: $x = a+b$

out(5) = in(5) \cup \{ a+b \}$

in(1) = \emptyset
in(2) = out(1)
in(3) = out(2) \cap out(5)
in(4) = out(3)
in(5) = out(4)

[x := a + b]^1;
[y := a*b]^2;
while [y > a + b]^3 {
  [a := a + 1]^4;
  [x := a + b]^5
}


Solution

\[ \text{in}(1) = \emptyset \]

1: \( x = a + b \)

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

2: \( y := a \times b \)

\[ \text{out}(2) = \{ a + b, a \times b \} \quad \text{in}(3) = \{ a + b \} \]

3: \( y > a + b \)

\[ \text{in}(4) = \text{out}(3) = \{ a + b \} \]

4: \( a = a + 1 \)

\[ \text{out}(4) = \emptyset \]

5: \( x = a + b \)

\[ \text{out}(5) = \{ a + b \} \]
Kill/Gen

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

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

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

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

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

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

\[ \text{in}(1) = \{(x,?), (y,?), (z,?)\} \]

1: \[ z = x+y \]

out(1) = ( \text{in}(1) \setminus \{(z,?)\} ) \cup \{(z,1)\}

\[ \text{in}(2) = \text{out}(1) \cup \text{out}(3) \]

2: true

out(2) = in(2)

\[ \text{in}(3) = \text{out}(2) \]

3: skip

out(3) = in(3)

After simplification: \[ \text{in}(2) = \text{in}(2) \cup \{(x,?), (y,?), (z,1)\} \]

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

\[ x := 2 \]
\[ y := 4 \]
\[ x := 1 \]
\[ \text{if } [y > x] \text{ then } [z := y] \]
\[ \text{else } [z := y \times y] \]
\[ x := z \]

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

\[ x := 2 \]
\[ y := 4 \]
\[ x := 1 \]
(if \( y > x \) then \( z := y \)
else \( z := y \times y \));
\[ x := z \]
Live Variables

\[ x := 2 \]
\[ y := 4 \]
\[ x := 1 \]
(if \[ y > x \] then \[ z := y \] else \[ z := y \times y \] );
\[ x := z \]

<table>
<thead>
<tr>
<th>Block</th>
<th>kill</th>
<th>gen</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ x := a ]</td>
<td>{ x }</td>
<td>{ FV(a) }</td>
</tr>
<tr>
<td>[ skip ]</td>
<td>Ø</td>
<td>Ø</td>
</tr>
<tr>
<td>[ b ]</td>
<td>Ø</td>
<td>FV(b)</td>
</tr>
</tbody>
</table>
[x := 2];
[y := 4];
[x := 1];
(if [y > x] then [z := y] else [z := y * y]);
[x := z]

<table>
<thead>
<tr>
<th>Block</th>
<th>kill</th>
<th>gen</th>
</tr>
</thead>
<tbody>
<tr>
<td>[x := a]</td>
<td>{ x }</td>
<td>{ FV(a) }</td>
</tr>
<tr>
<td>[skip]</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td>[b]</td>
<td>∅</td>
<td>FV(b)</td>
</tr>
</tbody>
</table>

Live Variables: solution
Why solution with smallest set?

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

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

\[
\text{In}(\text{lab}) = \begin{cases} 
\text{Initial} & \quad \text{when } \text{lab} \in \text{Entry labels} \\
\sqcup \{ \text{out}(\text{lab}') | (\text{lab}',\text{lab}) \in \text{CFG edges} \} & \quad \text{otherwise}
\end{cases}
\]

\[
\text{out}(\text{lab}) = f_{\text{lab}}(\text{in}(\text{lab}))
\]

- \(\sqcup\) is \(\cup\) or \(\sqcap\)
- CFG edges go either forward or backwards
- Entry labels are either initial program labels or final program labels (when going backwards)
- Initial is an initial state (or final when going backwards)
- \(f_{\text{lab}}\) is the transfer function associated with the blocks \(B_{\text{lab}}\)
Forward vs. Backward Analyses

1: \texttt{x := 2} \quad \{ (x,\texttt{?}), (y,\texttt{?}), (z,\texttt{?}) \}
2: \texttt{y := 4} \quad \{ (x,1), (y,\texttt{?}), (z,\texttt{?}) \}
4: y > x
5: z := y \quad \{ (x,1), (y,2), (z,\texttt{?}) \}
6: z = y*y
7: x := z

\{ (x,1), (y,2), (z,\texttt{?}) \}

\{ \texttt{z} \}

\{ \texttt{y} \}

\{ \texttt{y} \}

\{ \texttt{y} \}

\{ \texttt{z} \}

\{ \texttt{z} \}

\{ \texttt{z} \}
Must vs. May Analyses

- **When $\sqcup$ 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 $\sqcup$ 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

- $L = \emptyset (\text{Var} \times \text{Lab})$ is partially ordered by $\subseteq$
- $\uplus$ is $\bigcup$
- $L$ satisfies the Ascending Chain Condition because $\text{Var} \times \text{Lab}$ is finite (for a given program)
Example: Available Expressions

- $L = \emptyset (AExp)$ is partially ordered by $\supseteq$
- $\sqcap$ is $\cap$
- $L$ satisfies the Ascending Chain Condition because $AExp$ is finite (for a given program)
## Analyses Summary

<table>
<thead>
<tr>
<th></th>
<th>Reaching Definitions</th>
<th>Available Expressions</th>
<th>Live Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>ψ(Var × Lab)</td>
<td>ψ(AExp)</td>
<td>ψ(Var)</td>
</tr>
<tr>
<td>⊆</td>
<td>⊆</td>
<td>⊇</td>
<td>⊆</td>
</tr>
<tr>
<td>∪</td>
<td>∪</td>
<td>∩</td>
<td>∪</td>
</tr>
<tr>
<td>⊥</td>
<td>∅</td>
<td>AExp</td>
<td>∅</td>
</tr>
<tr>
<td>Initial</td>
<td>{ (x,?)</td>
<td>x ∈ Var}</td>
<td>∅</td>
</tr>
<tr>
<td>Entry labels</td>
<td>{ init }</td>
<td>{ init }</td>
<td>final</td>
</tr>
<tr>
<td>Direction</td>
<td>Forward</td>
<td>Forward</td>
<td>Backward</td>
</tr>
<tr>
<td>F</td>
<td>{ f: L → L</td>
<td>∃ k, g : f(val) = (val \ k) U g }</td>
<td></td>
</tr>
<tr>
<td>f_{lab}</td>
<td>f_{lab}(val) = (val \ kill) U gen</td>
<td></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 \( x \subseteq x' \) implies \( f(x) \subseteq f(x') \)
- Assume \( x \subseteq x' \), then for kill set \( k \) and gen set \( g \)
  \((x \setminus k) \cup g \subseteq (x' \setminus k) \cup g\)
- 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 $f(x \sqcup y) \subseteq f(x) \sqcup f(y)$

  - $f(x \sqcup y) = ((x \sqcup y) \setminus k) \cup g$
  - $= ((x \setminus k) \sqcap (y \setminus k)) \cup g$
  - $= (((x \setminus k) \cup g) \sqcap ((y \setminus k) \cup g))$
  - $= f(x) \sqcap f(y)$

- Used distributivity of $\sqcap$ and $\cup$
  - Works regardless of whether $\sqcup$ is $\cup$ or $\cap$
Points-to Analysis

- Many flavors
- PWHILE language

\[ p \in \text{PExp} \quad \text{pointer expressions} \]

\[ a ::= x | n | a_1 \text{ op } \ a_2 | \& x | * x | \text{nil} \]

\[ S ::= \lbrack x := a \rbrack_{\text{lab}} \]
\[ | \lbrack \text{skip} \rbrack_{\text{lab}} \]
\[ | S_1;S_2 \]
\[ | \text{if } [b]_{\text{lab}} \text{ then } S_1 \text{ else } S_2 \]
\[ | \text{while } [b]_{\text{lab}} \text{ do } S \]
\[ | x = \text{malloc} \]
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

\[ x := \&z \]
\[ y := \&z \]
\[ w := \&y \]
\[ r := w \]

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

- Property Space
  - $L = (\varnothing (\text{Var} \times \text{Var}), \subseteq, \cup, \cap, \emptyset, \text{Var} \times \text{Var})$

- Transfer functions

<table>
<thead>
<tr>
<th>Statement</th>
<th>out(lab)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[p = &amp;x]^{\text{lab}}$</td>
<td>in(lab) $\cup { (p,x) }$</td>
</tr>
<tr>
<td>$[p = q]^{\text{lab}}$</td>
<td>in(lab) $\cup { (p,x)</td>
</tr>
<tr>
<td>$[*p = q]^{\text{lab}}$</td>
<td>in(lab) $\cup { (r,x)</td>
</tr>
<tr>
<td>$[p = *q]^{\text{lab}}$</td>
<td>in(lab) $\cup { (p,r)</td>
</tr>
</tbody>
</table>
(May) Points-to Analysis

- What to do with malloc?
- Need some static naming scheme for dynamically allocated objects

- Single name for the entire heap
  - $\llbracket[p = \text{malloc}]^\text{lab}\rrbracket(S) = S \cup \{ (p,H) \}$

- Name based on static allocation site
  - $\llbracket[p = \text{malloc}]^\text{lab}\rrbracket(S) = S \cup \{ (p,\text{lab}) \}$
(May) Points-to Analysis

\[ \begin{align*}
[x := \text{malloc}]^1; & \quad \emptyset \\
[y := \text{malloc}]^2; & \quad \{ (x,H) \} \\
\text{(if } [x==y]^3 \text{ then} & \quad \{ (x,H), (y,H) \} \\
\quad [z := x]^4 & \quad \{ (x,H), (y,H) \} \\
\text{else} & \quad \{ (x,H), (y,H), (z,H) \} \\
\quad [z := y]^5 & \quad \{ (x,H), (y,H), (z,H) \} \\
) & \quad \{ (x,H), (y,H), (z,H) \}
\end{align*} \]

Single name \( H \) for the entire heap
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”
(May) Points-to Analysis

\[
\begin{align*}
[x := \text{malloc}]^1; & \quad // A1 \\
[y := \text{malloc}]^2; & \quad // A2 \\
(\text{if} \ [x==y]^3 \text{ then} & \quad \{ (x,A1) \} \\
\quad \quad [z := x]^4 & \quad \{ (x,A1), (y,A2) \} \\
\quad \quad \text{else} & \quad \{ (x,A1), (y,A2) \} \\
\quad \quad [z := y]^5 & \quad \{ (x,A1), (y,A2), (z,A1) \} \\
); & \quad \{ (x,A1), (y,A2), (z,A1), (z,A2) \}
\end{align*}
\]

Allocation-site based naming (using A_{lab} instead of just “lab” for clarity)
Weak Updates

<table>
<thead>
<tr>
<th>Statement</th>
<th>out(lab)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[p = &amp;x]^{\text{lab}}]</td>
<td>(\text{in(lab)} \cup {(p,x)})</td>
</tr>
<tr>
<td>[[p = q]^{\text{lab}}]</td>
<td>(\text{in(lab)} \cup {(p,x) \mid (q,x) \in \text{in(lab)}})</td>
</tr>
<tr>
<td>[[\ast p = q]^{\text{lab}}]</td>
<td>(\text{in(lab)} \cup {(r,x) \mid (q,x) \in \text{in(lab)} \text{ and } (p,r) \in \text{in(lab)}})</td>
</tr>
<tr>
<td>[[p = \ast q]^{\text{lab}}]</td>
<td>(\text{in(lab)} \cup {(p,r) \mid (q,x) \in \text{in(lab)} \text{ and } (x,r) \in \text{in(lab)}})</td>
</tr>
</tbody>
</table>

\[\text{x := malloc}^1; \quad \text{// A1}\]
\[\text{y := malloc}^2; \quad \text{// A2}\]
\[\text{z := x}^3;\]
\[\text{z := y}^4;\]

\[\emptyset\]
(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

```
x := malloc;  
y := malloc;  
(if x == y then  
  z := x  
else  
  z := y);  
```
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
int (*pf)(int);

int fun1(int x) {
    if (x < 10)
        return (*pf) (x+1); // C1
    else
        return x;
}

int fun2(int y) {
    pf = &fun1;
    return (*pf) (y); // C2
}

void main() {
    pf = &fun2;
    (*pf)(5); // C3
}
main() {
    for (i = 0; i < n; i++) {
        t1 = f(0); // C1
        t2 = f(42); // C2
        t3 = f(42); // C3
        X[i] = t1 + t2 + t3;
    }
}

int f(int v)
return (v+1);

context sensitivity

```
main() {
    for (i = 0; i < n; i++) {
        t1 = f(0);  // C1
        t2 = f(42); // C2
        t3 = f(42); // C3
        X[i] = t1 + t2 + t3;
    }
}

int f(int v)
return (v+1);
```
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

```c
main() { f(); f(); }
f() { g(); g(); }
g() { h(); h(); }
h() { ... }
```
main() {
    for (i = 0; i < n; i++) {
        t1 = f(0); // C1
        t2 = f(42); // C2
        t3 = f(42); // C3
        X[i] = t1 + t2 + t3;
    }
}

int f(int v)
    return (v+1);
}
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

- foo()
  - Call bar()
- bar()
  - Call bar()
- zoo()
  - Call bar()
Interprocedural Analysis

begin
proc p() is
    [x := a + 1]
end
[a=7]
[call p()]
[print x]
[a=9]
[call p()]
[print a]
end

- Extend language with begin/end and with \texttt{[call p()]_{clab}^{rlab}}
- Call label clab, and return label rlab
IVP: Interprocedural Valid Paths

- IVP: all paths with matching calls and returns
- And prefixes
Valid Paths
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 (i matched )i | ε
  - valid ::= valid (i matched | matched

  - paths can be defined by a regular expression
The Join-Over-Valid-Paths (JVP)

- $\text{vpaths}(n)$ all valid paths from program start to $n$
- $\text{JVP}[n] = \bigcap \{[e_1, e_2, \ldots, e] \text{ (initial)} \mid (e_1, e_2, \ldots, e) \in \text{vpaths}(n)\}$
- $\text{JVP} \subseteq \text{JFP}$
  - 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
   \[ x := a + 1 \]
end
a=7
[call p()]
[print x]
a=9
[call p()]
[print a]
end
begin\(^0\)
proc p() is\(^1\)
  if \([b]\)\(^2\) then (\(^3\)
    \([a := a - 1]\)\(^3\) \([x := o, a := 7]\)\(^4\)
    \[call p()\] \([x := -2a + 5]\)\(^5\)
  )\(^6\)
  \([x := -2a + 5]\)\(^7\)
end\(^8\)
\([a=7]\)\(^9\); \[call p()\] \([x := 0, a := 7]\)\(^10\)
\[call p()\] \([x := 0, a := 7]\)\(^11\)
\[print(x)\] \([x := 0, a := 0]\)\(^12\)
end\(^13\)
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() is
if [b] then (
[a := a - 1]  
[call p()]  
[a := a + 1]  
)  
[x := -2 * a + 5]
end

[a=7]; [call p()]  ; [print(x)]
end
begin

proc p() is
    if [b]
        [a := a - 1] [x := o, a := o]
        [call p()] [x := o, a := T]
    end
    [a := a + 1] [x := T, a := T]
    [x := -2*a + 5]
end

[read(a)] ; [call p()] ; [print(x)]
end

\lambda e. [x \mapsto -2e(a) + 5, a \mapsto e(a)]
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