How many ways can we do Reaching Definitions?

- Data Flow analysis
- Constraint based approach
- Abstract Interpretation
- Type and Effect Systems

Running Example

\[
\begin{align*}
y &:= x; \quad // 1 \\
z &:= 1; \quad // 2 \\
\text{while } y > 1 \text{ do } \{ \quad // 3 \\
& \quad z := z \times y; \quad // 4 \\
& \quad y := y - 1 \}; \quad // 5 \\
y &:= 0 \quad // 6
\end{align*}
\]

Data Flow Analysis

- \( RD_{\text{out}}(1) = RD_{\text{out}}(1) - \{ (y, 1) \text{ in Lab} \} \uplus \{ (y, 1) \} \)
- \( RD_{\text{out}}(2) = RD_{\text{out}}(2) - \{ (z, 1) \text{ in Lab} \} \uplus \{ (z, 2) \} \)
- ...
- \( RD_{\text{out}}(3) = RD_{\text{out}}(2) \uplus RD_{\text{out}}(5) \)
- ...

Reaching Definitions

Done many ways
Finding a solution

• Let F be this set of equations
• Find RD s.t. \( RD = F(RD) \)
  – find least such solution
  – what is a non-minimal solution?

Constraint based approach

• \( RD_{exit}(1) \supseteq RD_{entry}(1) - \{ (y, l) | l \text{ in Lab} \} \)
• \( RD_{exit}(1) \supseteq \{ (y, 1) \} \)
• \( RD_{exit}(2) \supseteq RD_{entry}(2) - \{ (z, l) | l \text{ in Lab} \} \)
• \( RD_{exit}(2) \supseteq \{ (z, 2) \} \)
• ...
• \( RD_{exit}(1) \supseteq \{ (x, ?), (y, ?), (z, ?) \} \)
• \( RD_{exit}(2) \supseteq RD_{entry}(1) \)
• \( RD_{exit}(3) \supseteq RD_{entry}(2) \)
• \( RD_{exit}(3) \supseteq RD_{entry}(5) \)
• ...

Something different

• \([ \{ \text{fn} \ x => [x] \} ^2 \ {\text{fn} \ y => [y]^3} ] ^4 \) ^5

• For each function application, which function may be applied?

• \( C(l) = \text{values that subexpression } l \text{ can evaluate to} \)
• \( p(x) = \text{values that } x \text{ can be bound to} \)
• \( [ \{ \text{fn} \ x => [x] \} ^2 \ {\text{fn} \ y => [y]^3} ] ^4 \) ^5
• \( \{ \text{fn} \ x => [x] \} \square C(2) \)
• \( \{ \text{fn} \ y => [y]^3 \} \square C(4) \)
Values taken on by a variable use

- $[[fn\ x\Rightarrow [x]]^{1}]^{2}\ [fn\ y\Rightarrow [y]]^{4}]^{5}$
- $p(x)\ni\ C(1)$
- $p(y)\ni\ C(3)$

Results of function application

- $[[fn\ x\Rightarrow [x]]^{1}]^{2}\ [fn\ y\Rightarrow [y]]^{4}]^{5}$
- Only one function application
  - but we don’t know what function will be applied
- If function might be $fn\ x\Rightarrow [x]^{1}$
  - $\{fn\ x\Rightarrow [x]^{1}\}\ni\ C(2)\ni\ C(4)\ni\ p(x)$
  - $\{fn\ x\Rightarrow [x]^{1}\}\ni\ C(2)\ni\ C(1)\ni\ C(5)$
- If function might be $fn\ y\Rightarrow [y]^{3}$
  - $\{fn\ y\Rightarrow [y]^{3}\}\ni\ C(2)\ni\ C(4)\ni\ p(y)$
  - $\{fn\ y\Rightarrow [y]^{3}\}\ni\ C(2)\ni\ C(3)\ni\ C(5)$

Least solution

- $[[fn\ x\Rightarrow [x]]^{1}]^{2}\ [fn\ y\Rightarrow [y]]^{4}]^{5}$
- $C(1) = \{fn\ y\Rightarrow [y]^{3}\}$
- $C(2) = \{fn\ x\Rightarrow [x]^{1}\}$
- $C(3) = \{\}$
- $C(4) = \{fn\ y\Rightarrow [y]^{3}\}$
- $C(5) = \{fn\ y\Rightarrow [y]^{3}\}$
- $p(x) = \{fn\ y\Rightarrow [y]^{3}\}$
- $p(y) = \{\}$

Abstract Interpretation

- A trace is a series of (label, variable) pairs indicating the sequence of assignments
- A collecting semantics records the set of traces $tr$ that can reach a given program point
- $RD(tr)(x) = l$ iff the rightmost pair $(x, l')$ in $tr$ has $l = l'$
Static Single Assignment Form

SSA Form
- Each variable is assigned to once
- Each use of a variable has exactly one reaching definition
- Makes a number of analysis techniques simpler and more effective

SSA of straight line code

Original
\begin{align*}
x &= 1 \\
y &= x+1 \\
x &= x+1 \\
z &= x+y
\end{align*}

In SSA form
\begin{align*}
x_1 &= 1 \\
y_1 &= x_1+1 \\
x_2 &= x_1+1 \\
z_1 &= x_2+y_1
\end{align*}

SSA with branches

Entry
\begin{align*}
x &= \text{read()} \\
\text{if } x > 0
\end{align*}

\begin{align*}
y &= -x \\
y &= \text{write(y)} \\
\text{Exit}
\end{align*}

Entry
\begin{align*}
x &= \text{read()} \\
\text{if } x > 0
\end{align*}

\begin{align*}
y_1 &= -x_1 \\
y &= x \\
y_2 &= \text{f}(y_1, y_1) \\
y_3 &= \text{write(y)} \\
\text{Exit}
\end{align*}
Phi functions

- Used when merging two values
- Phi functions are easy to handle in analysis
  - not so easy to execute
- To generate executable code, need to eliminate the phi functions

Why SSA?

- More efficient
  - avoids some quadratic cases:
    - programs with M defs and N uses can require M * N def-use chains
    - for almost all realistic programs, SSA is linear in size of program
- Makes algorithms more efficient and accurate
- Unrelated uses of the same variable name are irrelevant

SSA with loops

Where do we place Phi functions?

- An assignment to x in node B generates a new assignment/phi function for x in the dominance frontier of B
  - which may, in turn, introduce additional assignments/phi functions for x
**Dominance Frontier**

- C is in the dominance frontier of B iff
  - Exists a path from B to Exit through C
  - such that C is the first node not strictly dominated by B
- Equivalently:
  - C is the first node where a path from B to Exit and a path from Entry to Exit (not going through B) meet
- Equivalently:
  - B dominates a predecessor of C
  - B does not strictly dominate C

**Computing Dominance Frontier**

- Do in postorder on dominator tree:
  \[ \text{DF}(B) = \text{Succ}(B) - \text{Sdom}(B) \]
  foreach block C s.t. idom(C) = B do
  \[ \text{DF}(B) \cap = \text{DF}(C) - \text{Sdom}(B) \]
- Equivalently:
  \[ \text{DF}(B) = \text{Succ}(B) \]
  foreach block C s.t. idom(C) = B do
  \[ \text{DF}(B) \cap = \text{DF}(C) \]
  \[ \text{DF}(B) = \text{Sdom}(B) \]