# COSE312: Compilers

## Lecture 19 — Data-Flow Analysis (1)

Hakjoo Oh
2017 Spring

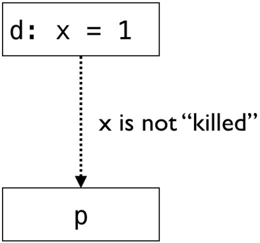
# Data-Flow Analysis

A collection of program analysis techniques that derive information about the flow of data along program execution paths, enabling safe code optimization, bug detection, etc.

- Reaching definitions analysis
- Live variables analysis
- Available expressions analysis
- Constant propagation analysis
- ...

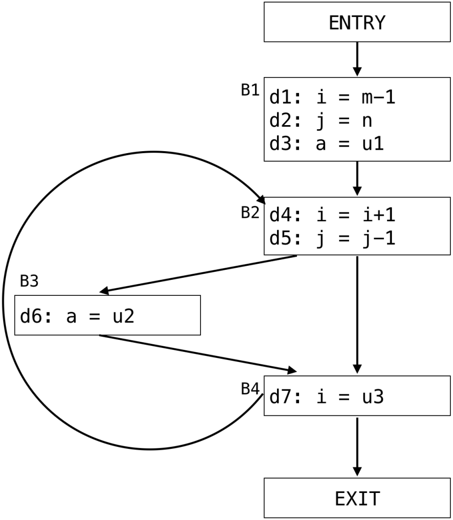
# Reaching Definitions Analysis

- A definition $d$ *reaches* a point $p$ if there is a path from the definition point to $p$ such that $d$ is not "killed" along that path.




- For each program point, RDA finds definitions that *can* reach the program point along some execution paths.

# Example: Reaching Definitions Analysis




IN(B1) = {}

OUT(B1) = {d1,d2,d3}

IN(B2) = {d1,d2,d3,d5,d6,d7}

OUT(B2) = {d3,d4,d5,d6}

IN(B3) = {d3,d4,d5,d6}

OUT(B3) = {d4,d5,d6}

IN(B4) = {d3,d4,d5,d6}

OUT(B4) = {d3,d5,d6,d7}

IN(EXIT) = {d3,d5,d6,d7}

OUT(EXIT) = {d3,d5,d6,d7}

# Applications

Reaching definitions analysis has many applications, e.g.,

- Simple constant propagation
  - For a use of variable $v$ in statement $n$: $\boxed{n : x = ...v...}$
  - If the definitions of $v$ that reach $n$ are all of the form $\boxed{d : v = c}$
  - Replace the use of $v$ in $n$ by $c$
- Uninitialized variable detection
  - Put a definition $\boxed{\text{d: x = any}}$ at the program entry.
  - For a use of variable $x$ in statement $n$: $\boxed{n : x = ...v...}$
  - If $d$ reaches $n$, $x$ is potentially uninitialized.
  - ```
    ...
    if (...) x = 1;
    ...
    a = x
    ```
- Loop optimization
  - If all of the reaching definitions of the operands of $n$ are outside of the loop, then $n$ can be moved out of the loop ("loop-invariant code motion")
  - `while (...) {...; n: z = x + y; ... }`
    ```

# The Analysis is Conservative

- Exact reaching definitions information cannot be obtained at compile time. It can be obtained only at runtime.
- ex) Deciding whether each path can be taken is undecidable:

```
a = rand(); b = rand(); c = rand();
if (a^10 + b^10 != c^10) {     // always true
  // (1)
} else {
  // (2)
}
```

- RDA computes an over-approximation of the reaching definitions that can be obtained at runtime.

# Reaching Definitions Analysis

The goal is to compute

$$\begin{array}{rcl}\textbf{in} & : & Block \rightarrow 2^{Definitions} \\ \textbf{out} & : & Block \rightarrow 2^{Definitions}\end{array}$$

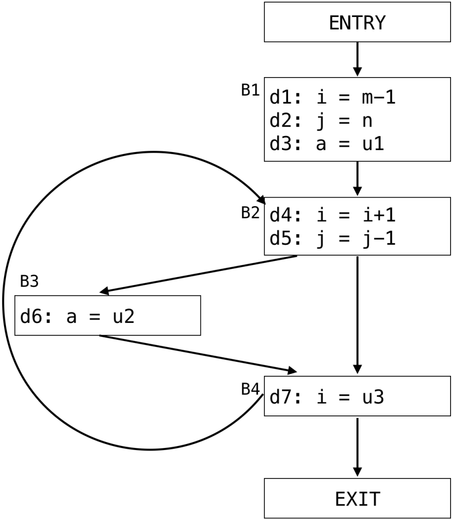
1. Compute gen/kill sets.
2. Derive transfer functions for each block in terms of gen/kill sets.
3. Derive the set of data-flow equations.
4. Solve the equation by the iterative fixed point algorithm.

# 1. Compute Gen/Kill Sets

$$\begin{array}{rcl} \textbf{gen} & : & Block \rightarrow 2^{Definitions} \\ \textbf{kill} & : & Block \rightarrow 2^{Definitions} \end{array}$$

- **gen**$(B)$: the set of definitions "generated" at block $B$
- **kill**$(B)$: the set of definitions "killed" at block $B$

# Example




gen(B1) =
kill(B1) =

gen(B2) =
kill(B2) =

gen(B3) =
kill(B3) =

gen(B4) =
kill(B4) =

# Exercise

Compute the **gen** and **kill** sets for the basic block $B$:

```
d1: a = 3
d2: a = 4
```

- $\textbf{gen}(B) =$
- $\textbf{kill}(B) =$

In general, when we have $k$ definitions in a block $B$:

```
d1; d2; ...; d_k
```

- $\textbf{gen}(B) =$
- $\textbf{kill}(B) =$

# 2. Transfer Functions

- The transfer function is defined for each basic block $B$:

$$f_B : 2^{Definitions} \rightarrow 2^{Definitions}$$

- The transfer function for a block $B$ encodes the semantics of the block $B$, i.e., how the block transfers the input to the output.

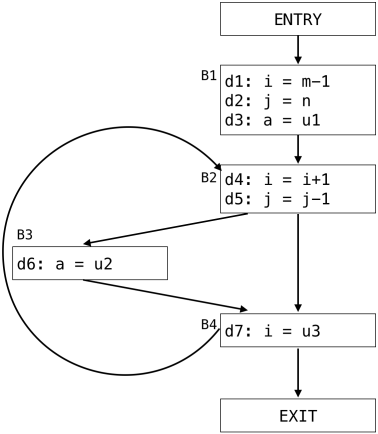
**B2**
```
d4: i = i+1      {d1,d2,d3,d5,d6,d7}
d5: j = j-1      {d3,d4,d5,d6}
```

- The semantics of $B$ is defined in terms of **gen**$(B)$ and **kill**$(B)$:

$$f_B(X) =$$

**B2**
```
d4: i = i+1      gen(B2) = {d4,d5}
d5: j = j-1      kill(B2) = {d1,d2,d7}
```

# 3. Derive Data-Flow Equations




$$
\begin{aligned}
\textbf{in}(B_1) &= \emptyset \\
\textbf{out}(B_1) &= f_{B_1}(\textbf{in}(B_1)) \\
\\
\textbf{in}(B_2) &= \textbf{out}(B_1) \cup \textbf{out}(B_4) \\
\textbf{out}(B_2) &= f_{B_2}(\textbf{in}(B_2)) \\
\\
\textbf{in}(B_3) &= \textbf{out}(B_2) \\
\textbf{out}(B_3) &= f_{B_3}(\textbf{in}(B_3)) \\
\\
\textbf{in}(B_4) &= \textbf{out}(B_2) \cup \textbf{out}(B_3) \\
\textbf{out}(B_4) &= f_{B_4}(\textbf{in}(B_4))
\end{aligned}
$$

# Data-Flow Equations

In general, the data-flow equations can be written as follows:

$$\begin{aligned}
\textbf{in}(B_i) &= \bigcup_{P \hookrightarrow B_i} \textbf{out}(P) \\
\textbf{out}(B_i) &= f_{B_i}(\textbf{in}(B_i)) \\
&= \textbf{gen}(B_i) \cup (\textbf{in}(B_i) - \textbf{kill}(B_i))
\end{aligned}$$

where $(\hookrightarrow)$ is the control-flow relation.

# 4. Solve the Equations

- The desired solution is the *least* **in** and **out** that satisfies the equations (why least?):

$$\begin{array}{rcl} \mathbf{in}(B_i) & = & \bigcup_{P \hookrightarrow B_i} \mathbf{out}(P) \\ \mathbf{out}(B_i) & = & \mathbf{gen}(B_i) \cup (\mathbf{in}(B_i) - \mathbf{kill}(B_i)) \end{array}$$

- The solution is defined as $\mathit{fix} F$, where $F$ is defined as follows:

$$F(\mathbf{in}, \mathbf{out}) = (\lambda B. \bigcup_{P \hookrightarrow B} \mathbf{out}(P), \lambda B. f_B(\mathbf{in}(B)))$$

The least fixed point $\mathit{fix} F$ is computed by

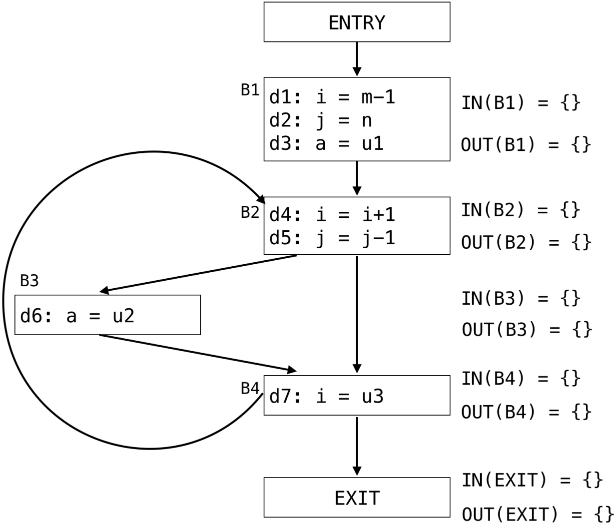
$$\bigcup_{i \geq 0} F^i(\lambda B.\emptyset, \lambda B.\emptyset)$$

# The Fixpoint Algorithm

The equations are solved by the iterative fixed point algorithm:

For all $i$, $\mathbf{in}(B_i) = \mathbf{out}(B_i) = \emptyset$
**while** (changes to any **in** and **out** occur) {
For all $i$, update
$\mathbf{in}(B_i) = \bigcup_{P \hookrightarrow B_i} \mathbf{out}(P)$
$\mathbf{out}(B_i) = \mathbf{gen}(B_i) \cup (\mathbf{in}(B_i) - \mathbf{kill}(B_i))$
}

# Example


ENTRY
B1
d1: i = m-1
d2: j = n
d3: a = u1
IN(B1) = {}
OUT(B1) = {}
B2
d4: i = i+1
d5: j = j-1
IN(B2) = {}
OUT(B2) = {}
B3
d6: a = u2
IN(B3) = {}
OUT(B3) = {}
B4
d7: i = u3
IN(B4) = {}
OUT(B4) = {}
EXIT
IN(EXIT) = {}
OUT(EXIT) = {}

# Summary

- Code optimization requires static analysis, data-flow analysis.
- Every static analysis follows two steps:
  1. Set up a set of *abstract semantic equations*.
     - about dynamics of program executions (e.g., how definitions flow)
  2. Solve the equations using the iterative fixed point algorithm.
     - naive tabulation algorithm, worklist algorithm, etc