EC3204: Programming Languages and Compilers

Lecture 17 — Optimization (2)
Data-Flow Analysis

Sunbeom So
Fall 2024

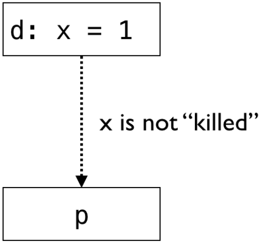
Data-Flow Analysis

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

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

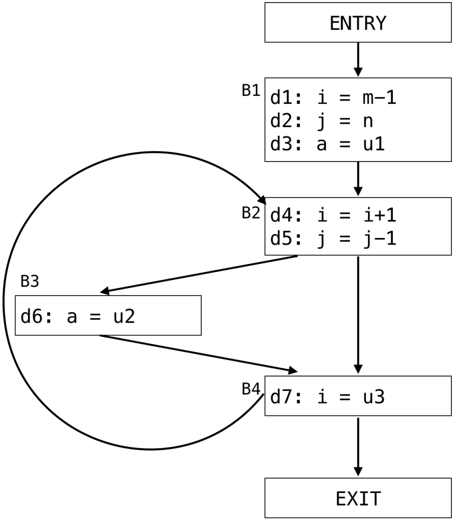
Reaching Definitions Analysis

- A definition d is a program point where the value of a variable may be updated.
- A definition d **reaches** a point p if there is a path from d to p such that d is not "killed" along that path (i.e., a defined variable in d is not redefined along that path).



- For each program point p, RDA aims to find definitions that **may** reach p along possible execution paths.

Example: Reaching Definitions Analysis

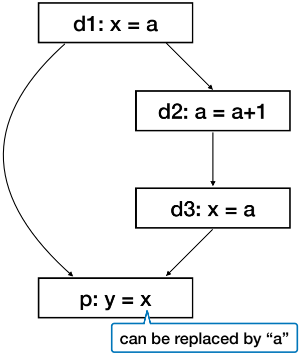


```
d1: i = m-1
d2: j = n
d3: a = u1

d4: i = i+1
d5: j = j-1

d6: a = u2

d7: i = u3
```

```
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}
```

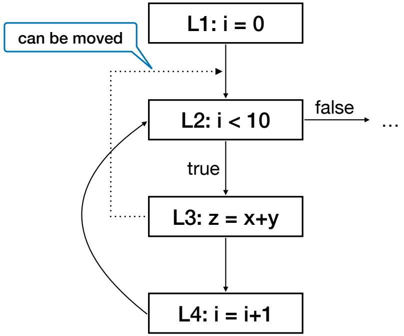
Application Example (1): Copy Propagation

- Suppose some variable x is used in a program point p, all the statements in its definitions, $\{d_1, \cdots, d_n\}$, are of the form $x = a$.
- Then, we can safely replace x with a at the program point p.



Application Example (2): Loop Optimization

Given some used variables at a program point p, if all of their reaching definitions are outside of the loop, then the instruction at p can be moved out of the loop (loop-invariant code motion).



Reaching Definitions Analysis Must Be Conservative

- Since deciding whether each program path can be taken or not is undecidable (halting problem), exact reaching definitions cannot be computed mechanically.
- For compiler optimizations, RDA computes a safe approximation (over-approximation) of the reaching definitions that can be obtained at runtime.

Q. What happens if we fail to overapproximate the reaching definitions?

Reaching Definitions Analysis Must Be Conservative

- Since deciding whether each program path can be taken or not is undecidable (halting problem), exact reaching definitions cannot be computed mechanically.
- For compiler optimizations, RDA computes a safe approximation (over-approximation) of the reaching definitions that can be obtained at runtime.

Q. What happens if we fail to overapproximate the reaching definitions?
A. May perform unsafe code transformation.

Goal of Reaching Definitions Analysis

The goal is to compute the set of definitions valid at the entry and exit of each block:

$$\begin{aligned} \textbf{in} &: \quad \mathit{Block} \rightarrow 2^{\mathit{Definitions}} \\ \textbf{out} &: \quad \mathit{Block} \rightarrow 2^{\mathit{Definitions}} \end{aligned}$$

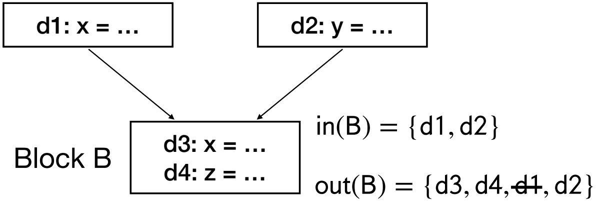
Reaching definitions can be computed by following the general data-flow analysis recipe below.

1. Set up the set of data-flow equations.
2. Solve the equations by the iterative fixed point algorithm.

Step 1. Set Up Data-Flow Equations

Intuitions:

1. The incoming definitions should contain definitions from every immediate preceding block.
2. The outgoing definitions should contain definitions generated at a block.
3. The outgoing definitions should preserve incoming definitions, except for definitions killed at the block.



Step 1. Set Up Data-Flow Equations

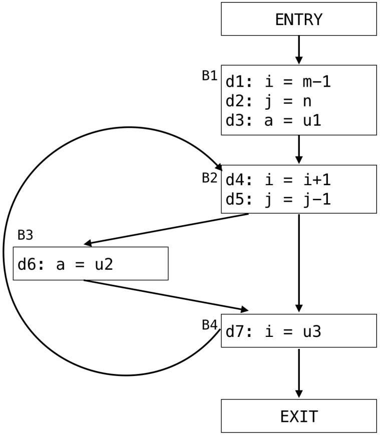
In general, the data-flow equations can be written as follows:

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

where

- $\textbf{pred}(B_i) = \{B \mid B \hookrightarrow B_i\}$ is the set of B_i's predecessors and $(\hookrightarrow)$ is the control-flow relation.
- f_B is a transfer function for each block B (an abstract semantic function for B).
- $\textbf{gen}(B)$: the set of definitions "generated" at block B.
- $\textbf{kill}(B)$: the set of definitions "killed" at block B.

Example



$$\begin{aligned}
\textbf{in}(B_1) &= \emptyset \\
\textbf{out}(B_1) &= f_B(\textbf{in}(B_1)) \\[1ex]
\textbf{in}(B_2) &= \textbf{out}(B_1) \cup \textbf{out}(B_4) \\
\textbf{out}(B_2) &= f_B(\textbf{in}(B_2)) \\[1ex]
\textbf{in}(B_3) &= \textbf{out}(B_2) \\
\textbf{out}(B_3) &= f_B(\textbf{in}(B_3)) \\[1ex]
\textbf{in}(B_4) &= \textbf{out}(B_2) \cup \textbf{out}(B_3) \\
\textbf{out}(B_4) &= f_B(\textbf{in}(B_4))
\end{aligned}$$

Computing **gen** and **kill** Sets

Suppose we have k definitions $(d1; d2; \cdots; d_k)$ in a block B.

- **gen**(B): the set of definitions "generated" at block B.

$$\begin{aligned}\textbf{gen}(B) \;=\; & \textbf{gen}(d_k) \\ & \cup\ (\textbf{gen}(d_{k-1}) - \textbf{kill}(d_k)) \\ & \cup\ (\textbf{gen}(d_{k-2}) - \textbf{kill}(d_{k-1}) - \textbf{kill}(d_k)) \\ & \cdots \\ & \cup\ (\textbf{gen}(d_1) - \textbf{kill}(d_2) - \textbf{kill}(d_3) - \cdots - \textbf{kill}(d_k))\end{aligned}$$

  where **gen**$(d_i) = \{d_i\}$.

- **kill**(B): the set of definitions "killed" at block B.

$$\textbf{kill}(B) = \textbf{kill}(d_1) \cup \textbf{kill}(d_2) \cup \cdots \cup \textbf{kill}(d_k)$$

  where **kill**(d_i) is the set of all other definitions that define the same variable.

Example

Consider the block B $\boxed{\texttt{d1: a = 3; d2: a = 4}}$. Assume the variable a is not defined in the other blocks.

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

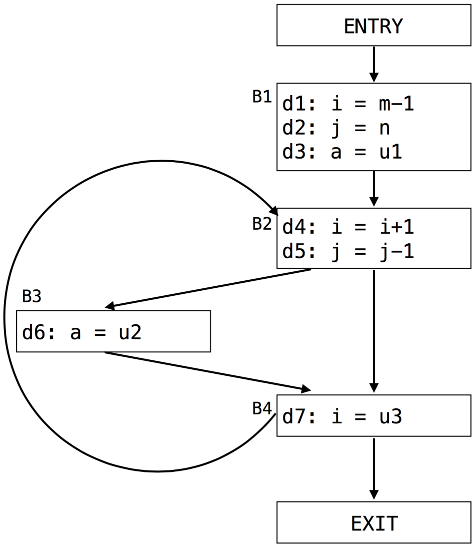
Example

Consider the block B $\boxed{\texttt{d1: a = 3; d2: a = 4}}$. Assume the variable a is not defined in the other blocks.

- $\textbf{gen}(B) = \{\texttt{d2}\}$
- $\textbf{kill}(B) = \{\texttt{d1}, \texttt{d2}\}$

d_1 kills d_2, and vice versa.

Example



```
gen(B1) = {d1,d2,d3}
kill(B1) = {d4,d5,d6,d7}

gen(B2) = {d4,d5}
kill(B2) = {d1,d2,d7}

gen(B3) = {d6}
kill(B3) = {d3}

gen(B4) = {d7}
kill(B4) = {d1,d4}
```

Step 2. Solve the Data-Flow Equations

- The desired solution is the the least **in** and **out** that satisfies the equations:

$$\mathbf{in}(B_i) = \bigcup_{P \in \mathbf{pred}(B_i)} \mathbf{out}(P)$$

$$\mathbf{out}(B_i) = \mathbf{gen}(B_i) \cup (\mathbf{in}(B_i) - \mathbf{kill}(B_i))$$

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

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

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

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

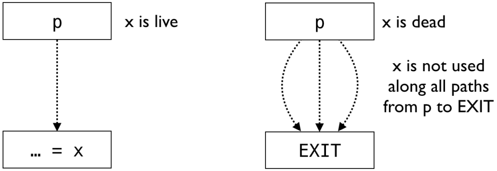
Step 2. Solve the Data-Flow Equations

The equations are solved by the iterative fixed point algorithm.

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

Liveness Analysis

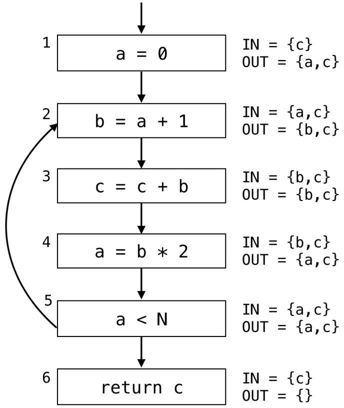
- A variable is **live** at program point p iff its value could be used in the future (i.e., at some subsequent point along paths starting at p).



- Liveness analysis aims to compute the set of live variables for each basic block of the program.

Example: Liveness of Variables

Unlike reaching definitions analysis, liveness is computed backwardly.



Applications

- Deadcode elimination (removing assignments whose assigned variables never get used)
  - Suppose we have a statement: $\boxed{\text{n: x = y + z}}$
  - We can eliminate the statement if x is dead (i.e., x is not included in the live variable set at n).
- Uninitialized variable detection
  - Any variables live at the program entry are potentially uninitialized, except for parameters.
- Register allocation
  - If two variables a and b never live at the same time, the same register can be assigned to them.

Goal of Liveness Analysis

The goal is to compute the set of live variables at the entry and exit of each block:

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

1. Set up the set of data-flow equations.
2. Solve the equations by the iterative fixed point algorithm.

Step 1. Set up Data-Flow Equations

Intuitions:

1. The live variables at the exit should contain variables from every immediate following block.
2. The live variables at the entry should contain variables used in B, denoted **use**(B).
3. The live variables at the entry should preserve variables from outside, except for variables defined in B, denoted **def**(B).

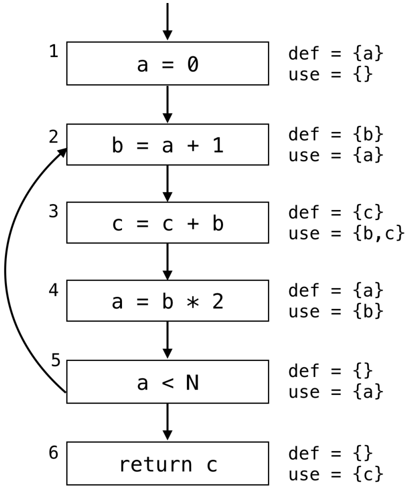
Equations:

$$\begin{aligned} \textbf{in}(B) &= \textbf{use}(B) \cup (\textbf{out}(B) - \textbf{def}(B)) \\ \textbf{out}(B) &= \bigcup_{S \in \textbf{succ}(B)} \textbf{in}(S) \end{aligned}$$

where $\textbf{succ}(B) = \{S \mid B \hookrightarrow S\}$ is the set of B's successors.

Example: **Def** and **Use** Sets

Straightforward as we do not consider variable aliasing in this course.

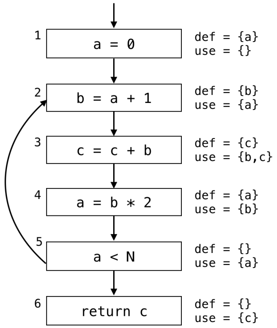


Step 2. Solve the Data-Flow Equations

The equations are solved by the iterative fixed point algorithm.

```
For all $i$, $\textbf{in}(B_i) = \textbf{out}(B_i) = \emptyset$
**while** (changes to any $\textbf{in}(B_i)$ and $\textbf{out}(B_i)$ occur) {
  For all $i$, update
    $\textbf{in}(B_i) = \textbf{use}(B_i) \cup (\textbf{out}(B_i) - \textbf{def}(B_i))$
    $\textbf{out}(B_i) = \bigcup_{S \in \textbf{succ}(B_i)} \textbf{in}(S)$
}
```

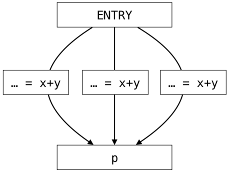
Example



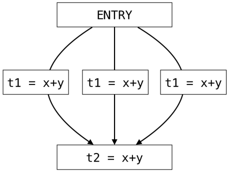
			1st		2nd		3rd	
	use	def	out	in	out	in	out	in
6	$\{c\}$	$\emptyset$	$\emptyset$	$\{c\}$	$\emptyset$	$\{c\}$	$\emptyset$	$\{c\}$
5	$\{a\}$	$\emptyset$	$\{c\}$	$\{a,c\}$	$\{a,c\}$	$\{a,c\}$	$\{a,c\}$	$\{a,c\}$
4	$\{b\}$	$\{a\}$	$\{a,c\}$	$\{b,c\}$	$\{a,c\}$	$\{b,c\}$	$\{a,c\}$	$\{b,c\}$
3	$\{b,c\}$	$\{c\}$	$\{b,c\}$	$\{b,c\}$	$\{b,c\}$	$\{b,c\}$	$\{b,c\}$	$\{b,c\}$
2	$\{a\}$	$\{b\}$	$\{b,c\}$	$\{a,c\}$	$\{b,c\}$	$\{a,c\}$	$\{b,c\}$	$\{a,c\}$
1	$\emptyset$	$\{a\}$	$\{a,c\}$	$\{c\}$	$\{a,c\}$	$\{c\}$	$\{a,c\}$	$\{c\}$

Available Expressions Analysis

- An expression $x + y$ is **available** at a point p iff every path from the entry node to p evaluates $x + y$, and after the last such evaluation prior to reaching p, there are no subsequent assignments to x or y.



- Application: common subexpression elimination (eliminating duplicated computations of an expression e)



Goal of Available Expressions Analysis

The goal is to compute the set of available expressions at the entry and exit of each block.

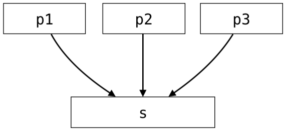
$$\begin{aligned} \textbf{in} &: \mathit{Block} \to 2^{\mathit{Expr}} \\ \textbf{out} &: \mathit{Block} \to 2^{\mathit{Expr}} \end{aligned}$$

1. Derive the set of data-flow equations.
2. Solve the equations by the iterative fixed point algorithm.

Step 1. Set Up Data-Flow Equations

Intuitions:

1. At the program entry, no expressions are available.
2. An expression is available at the entry only if it is available at the end of all its predecessors.
3. The available expressions at the exit should contain expressions that appeared in a block.
4. The available expressions at the exit should preserve the incoming expressions, except for expressions that may be invalidated in the block.



Step 1. Set Up Data-Flow Equations

Equations:

$$\begin{aligned}
\textbf{in}(ENTRY) &= \emptyset \\
\textbf{in}(B) &= \bigcap_{P \in \textbf{pred}(B)} \textbf{out}(P) \\
\textbf{out}(B) &= \textbf{gen}(B) \cup (\textbf{in}(B) - \textbf{kill}(B))
\end{aligned}$$

gen and **kill** sets

- **gen**(B): the set of expressions evaluated and not subsequently killed
- **kill**(B): the set of expressions whose variables can be killed and that are not subsequently recomputed
- What expressions are generated and killed by each of statements?

Statement s	**gen**(s)	**kill**(s)
$x = y + z$	$\{y+z\} - \textbf{kill}(s)$	expressions containing x
$x = \texttt{alloc}(n)$	$\emptyset$	expressions containing x
$x = y[i]$	$\{y[i]\} - \textbf{kill}(s)$	expressions containing x
$x[i] = y$	$\emptyset$	expressions of the form $x[k]$

  Basically, $x = y + z$ generates $y + z$, but $y = y + z$ does not because y is subsequently killed.
- Q. What expressions are generated and killed by the block?

$$\boxed{\begin{array}{l} a = b + c \\ b = a - d \\ c = b + c \\ d = a - d \end{array}}$$

Step 2. Solve the Data-Flow Equations

- We are interested in the largest set satisfying the equation.
- That is, we need to find the greatest solution (i.e., greatest fixed point) of the equation.

```
$\textbf{in}(\mathit{ENTRY}) = \emptyset$
For other $B_i$, $\textbf{in}(B_i) = \textbf{out}(B_i) = \mathit{Expr}$
**while** (changes to any **in** and **out** occur) {
  For all $i$, update
    $\textbf{in}(B_i) = \bigcap_{P \in \textbf{pred}(B_i)} \textbf{out}(P)$
    $\textbf{out}(B_i) = \textbf{gen}(B_i) \cup (\textbf{in}(B_i) - \textbf{kill}(B_i))$
}
```