AAA616: Program Analysis

Lecture 10 — Data-Flow Analysis

Hakjoo Oh
2022 Fall

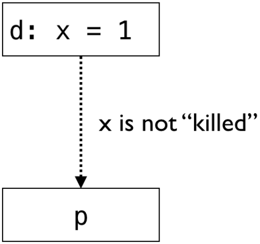
Data-Flow Analysis

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
- ...

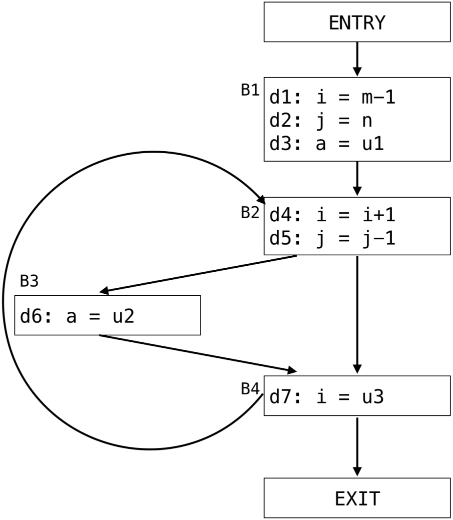
Reaching Definitions Analysis

- 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 $\boxed{\text{d: x} = \text{any}}$ 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; ... }`
    ```

Reaching Definitions Analysis

The goal is to compute

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

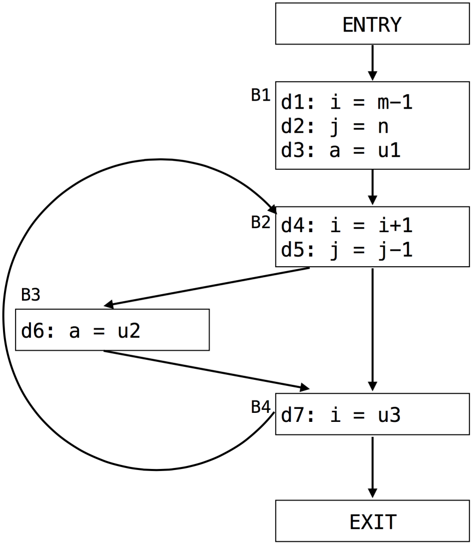
1. Compute gen/kill sets.
2. Derive transfer functions for each block in terms of gen/kill sets.
3. Derive the set of data-flow equations.
4. Solve the equation by the iterative fixed point algorithm.

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) = {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}
```

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
```

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

2. Transfer Functions

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

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

- 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.

$\boldsymbol{B2}$

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

```
{d1,d2,d3,d5,d6,d7}
{d3,d4,d5,d6}
```

- The semantics of B is defined in terms of $\mathbf{gen}(B)$ and $\mathbf{kill}(B)$:

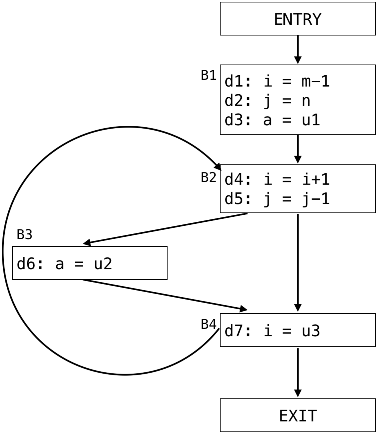
$$f_B(X) = \mathbf{gen}(X) \cup (X - \mathbf{kill}(X))$$

$\boldsymbol{B2}$

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

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

3. Derive Data-Flow Equations



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

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

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

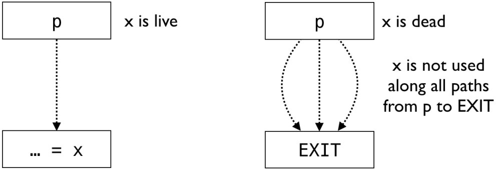
The Fixpoint Algorithm

The equations are solved by the iterative fixed point algorithm:

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

Liveness Analysis

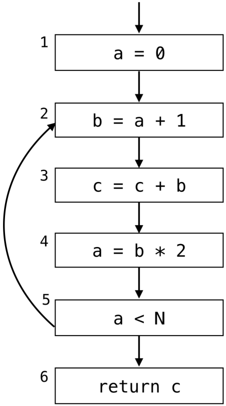
- A variable is *live* at program point p if its value could be used in the future (along some path starting at p).



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

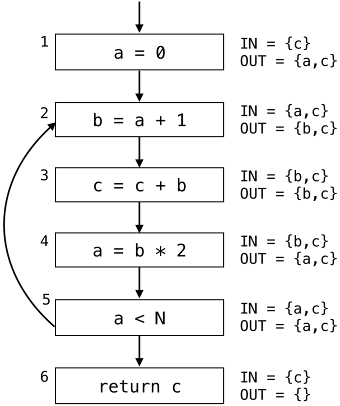
Example: Liveness of Variables

We analyze liveness from the future to the past.



- The live range of b: $\{2 \to 3, 3 \to 4\}$
- The live range of a: $\{1 \to 2, 4 \to 5 \to 2\}$ (not from $2 \to 3 \to 4$)
- The live range of c: the entire code

Example: Liveness of Variables



Applications

- Deadcode elimination
  - Problem: Eliminate assignments whose computed values never get used.
  - Solution: How?
  - Suppose we have a statement: $\boxed{\text{n: x} = \text{y} + \text{z}}$.
  - When x is dead at n, we can eliminate n.
- Uninitialized variable detection
  - Problem: Detect uninitialized use of variables
  - Solution: How? Any variables live at the program entry (except for parameters) are potentially uninitialized
- Register allocation
  - Problem: Rewrite the intermediate code to use no more temporaries than there are machine registers
  - Example:

```
a := c + d          r1 := r2 + r3
e := a + b          r1 := r1 + r4
f := e - 1          r1 := r1 - 1
```

  - Solution: How? Compute live ranges of variables. If two variables a and b never live at the same time, assign the same register to them.

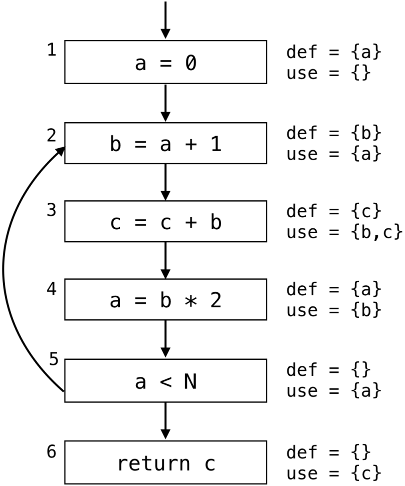
Liveness Analysis

The goal is to compute

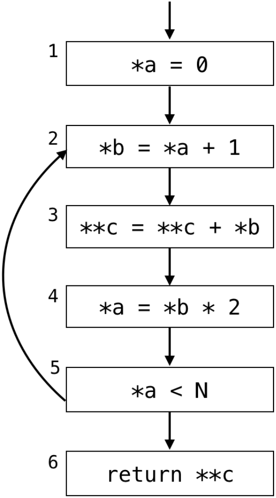
$$\begin{array}{rcl} \textbf{in} & : & Block \rightarrow 2^{Var} \\ \textbf{out} & : & Block \rightarrow 2^{Var} \end{array}$$

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

Def/Use Sets


1
a = 0
def = {a}
use = {}
2
b = a + 1
def = {b}
use = {a}
3
c = c + b
def = {c}
use = {b,c}
4
a = b * 2
def = {a}
use = {b}
5
a < N
def = {}
use = {a}
6
return c
def = {}
use = {c}

cf) Def/Use sets are only dynamically computable



Data-Flow Equations

Intuitions:

1. If a variable is in **use**(B), then it is live on entry to block B.
2. If a variable is live at the end of block B, and not in **def**(B), then the variable is also live on entry to B.
3. If a variable is live on enty to block B, then it is live at the end of predecessors of B.

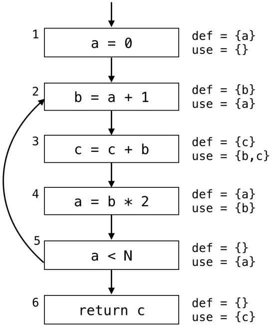
Equations:

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

Fixed Point Computation

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

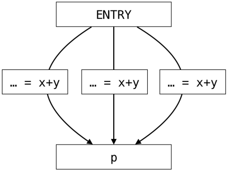
Example



			$1st$		$2nd$		$3rd$	
	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	$\emptyset$	$\{a\}$	$\{a,c\}$	$\{c\}$	$\{a,c\}$	$\{c\}$	$\{a,c\}$	$\{c\}$

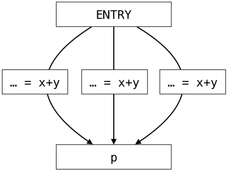
Available Expressions Analysis

- An expression $x + y$ is *available* at a point p if 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.

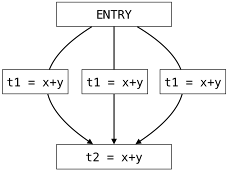


Available Expressions Analysis

- An expression $x + y$ is *available* at a point p if 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 (i.e., given a program that computes e more than once, eliminate one of the duplicate computations)



Available Expressions Analysis

The goal is to compute

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

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

Gen/Kill Sets

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

Statement s	**gen**(s)	**kill**(s)
$x = y + z$	$\{y + z\}$ − **kill**(s)	expressions containing x
$x = \texttt{alloc}(n)$	$\emptyset$	expressions containing x
$x = y[i]$	$\{y[i]\}$ − **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.

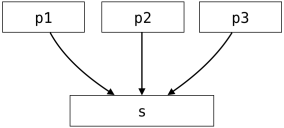
- What expressions are generated and killed by the block?

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

1. Set up a set of data-flow equations

Intuitions:

1. At the entry, no expressions are available.
2. An expression is available at the entry of a block only if it is available at the end of *all* its predecessors.



Equations:

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

2. Solve the equations

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

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

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