

COSE312: Compilers

Lecture 16 — Optimization

Hakjoo Oh
2026 Spring

Middle End: Optimizer

Converts the source program into a more efficient yet semantically equivalent program.



ex)

```
t1 = 10
t2 = rate * t1
t3 = init + t2
pos = t3
```

original IR

```
t1 = 10
t2 = rate * 10
t3 = init + t2
pos = t3
```

```
t2 = rate * 10
t3 = init + t2
pos = t3
```

```
t2 = rate * 10
pos = init + t2
```

final IR

Common Optimization Passes

- Common subexpressions elimination
- Copy propagation
- Deadcode elimination
- Constant folding
- ...

Common Subexpression Elimination

- An occurrence of an expression E is called a *common subexpression* if E was previously computed and the values of the variables in E have not changed since the previous computation.

```
x = 2 * k + 1
...           // no defs to k
y = 2 * k + 1
```

- We can avoid recomputing E by replacing E by the variable that holds the previous value of E .

```
x = 2 * k + 1
...           // no defs to k
y = x
```

Copy Propagation

After the copy statement $u = v$, use v for u unless u is re-defined.

$u = v$		$u = v$
$x = u + 1$		$x = v + 1$
$u = x$	\Rightarrow	$u = x$
$y = u + 2$		$y = x + 2$

cf) Copy instructions can be generated during optimization, e.g., common subexpression elimination.

Deadcode Elimination

- A variable is *live* at a point in a program if its value is used eventually; otherwise it is *dead* at that point.
- A statement is said to be *deadcode* if it computes values that never get used.

```
u = v      // deadcode
```

```
x = v + 1
```

```
u = x.     // deadcode
```

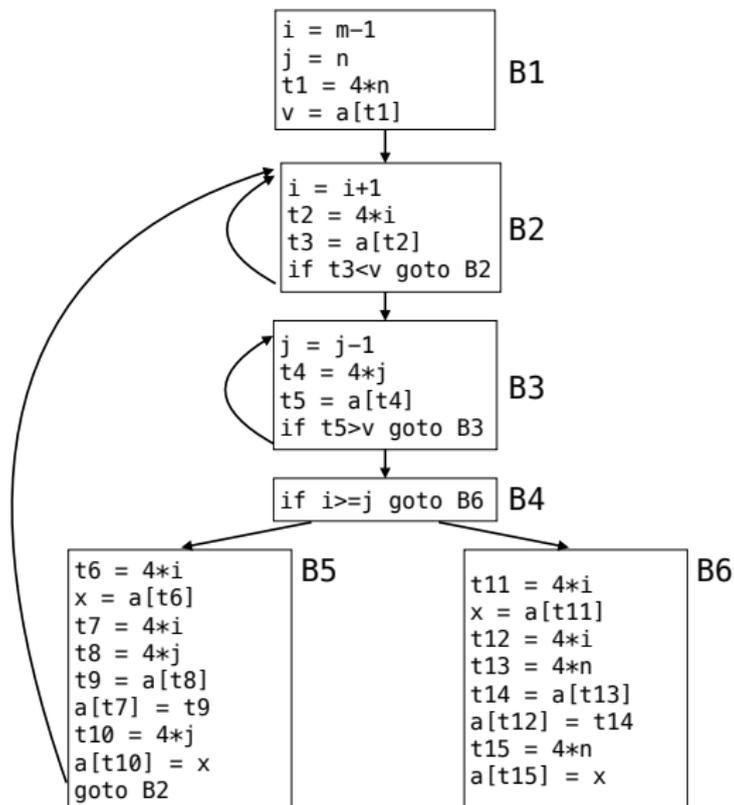
```
y = x + 2
```

Constant Folding

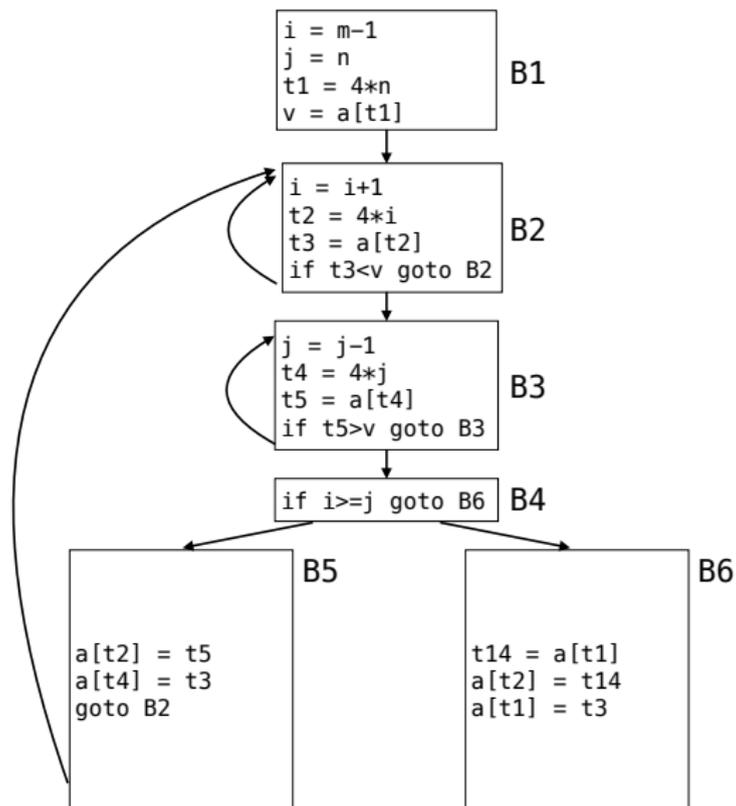
Decide that the value of an expression is a constant and use the constant instead.

$$\begin{array}{lcl} c = 1 & & c = 1 \\ x = c + c & \Rightarrow & x = 2 \\ y = x + x & & y = 4 \end{array}$$

Example: Original Program



Example: Optimized Program



Static analysis is needed

To optimize a program, we need static analysis that derives information about the flow of data along program execution paths. Examples:

- Do the two textually identical expressions evaluate to the same value along any possible execution path of the program? (If so, we can apply common subexpression elimination)
- Is the result of an assignment not used along any subsequent execution path? (If so, we can apply deadcode elimination).

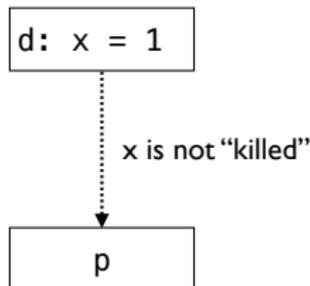
Data-Flow Analysis

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
- ...

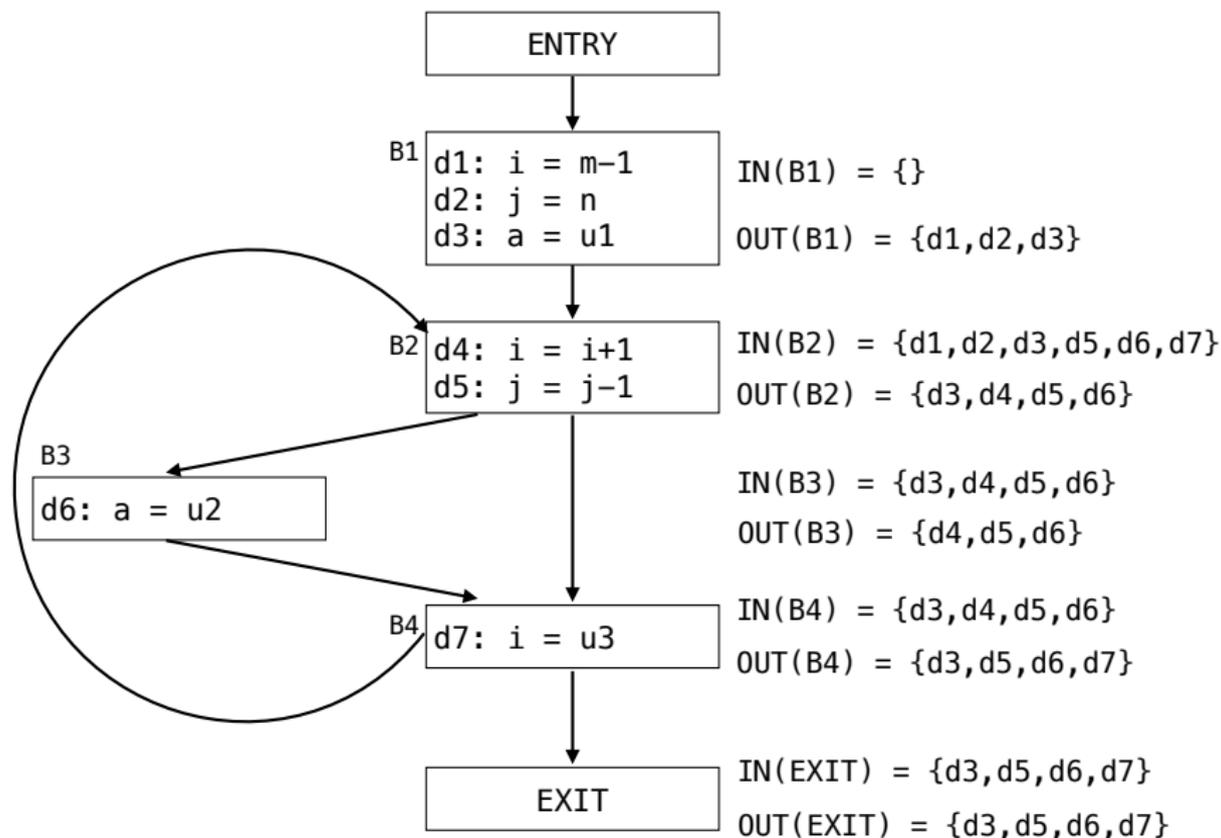
Reaching Definitions Analysis

- 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



Applications

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

- Simple constant propagation

- ▶ For a use of variable v in statement n : $n : x = \dots v \dots$
- ▶ If the definitions of v that reach n are all of the form $d : v = c$
- ▶ Replace the use of v in n by c

- Uninitialized variable detection

- ▶ Put a definition $d : x = \text{any}$ at the program entry.
- ▶ For a use of variable x in statement n : $n : v = \dots x \dots$
- ▶ 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(); k = rand();  
if (ak + bk != ck) {      // 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

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

- 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.

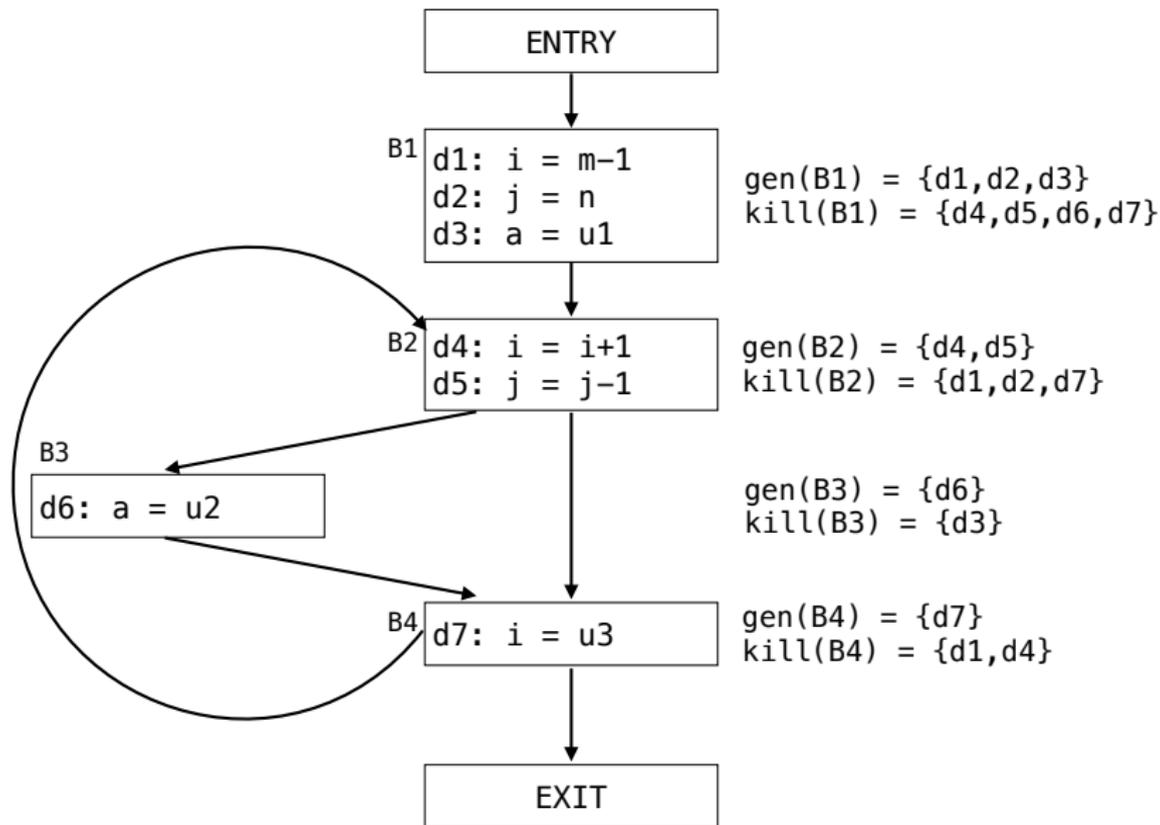
1. Compute Gen/Kill Sets

gen : *Block* $\rightarrow 2^{\text{Definitions}}$

kill : *Block* $\rightarrow 2^{\text{Definitions}}$

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

Example



Exercise

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

d1: a = 3

d2: a = 4

- $\text{gen}(B) = \{d_2\}$
- $\text{kill}(B) = \{d_1, d_2, \dots\}$

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

d1; d2; ...; d_k

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

2. Transfer Functions

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

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

- 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.

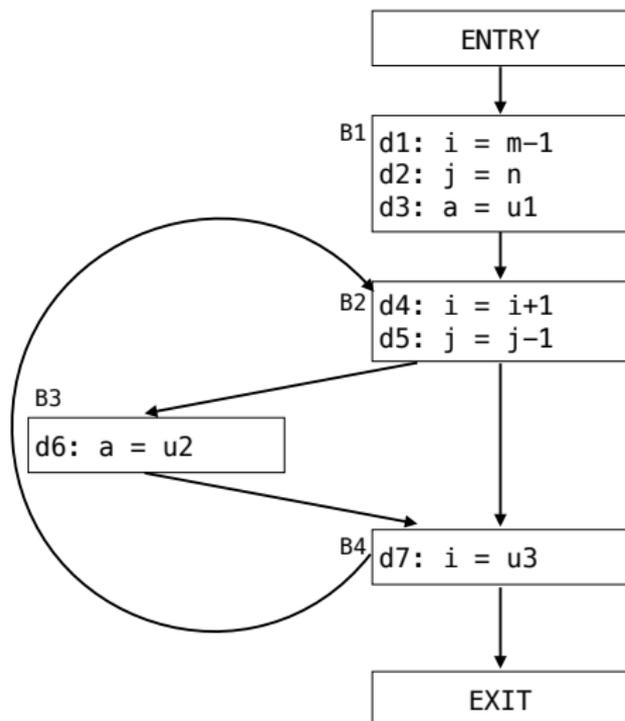
$$B2 \quad \boxed{\begin{array}{l} d4: i = i+1 \\ d5: j = j-1 \end{array}} \quad \begin{array}{l} \{d1, d2, d3, d5, d6, d7\} \\ \{d3, d4, d5, d6\} \end{array}$$

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

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

$$B2 \quad \boxed{\begin{array}{l} d4: i = i+1 \\ d5: j = j-1 \end{array}} \quad \begin{array}{l} \mathbf{gen}(B2) = \{d4, d5\} \\ \mathbf{kill}(B2) = \{d1, d2, d7\} \end{array}$$

3. Derive Data-Flow Equations



$$\begin{aligned}\mathbf{in}(B_1) &= \emptyset \\ \mathbf{out}(B_1) &= f_{B_1}(\mathbf{in}(B_1))\end{aligned}$$

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

$$\begin{aligned}\mathbf{in}(B_3) &= \mathbf{out}(B_2) \\ \mathbf{out}(B_3) &= f_{B_3}(\mathbf{in}(B_3))\end{aligned}$$

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

Data-Flow Equations

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

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

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?):

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

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

$$F : (\mathit{Block} \rightarrow 2^{\mathit{Definitions}})^2 \rightarrow (\mathit{Block} \rightarrow 2^{\mathit{Definitions}})^2$$

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

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

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

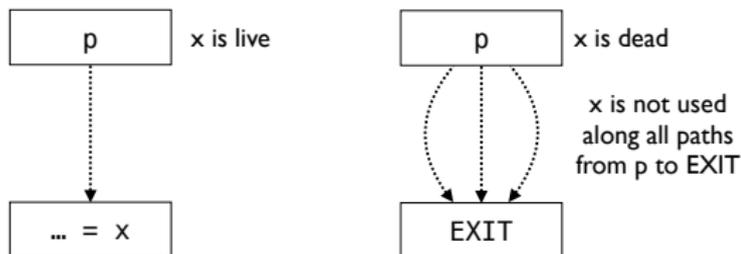
Fixed Point Algorithm

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

Liveness Analysis

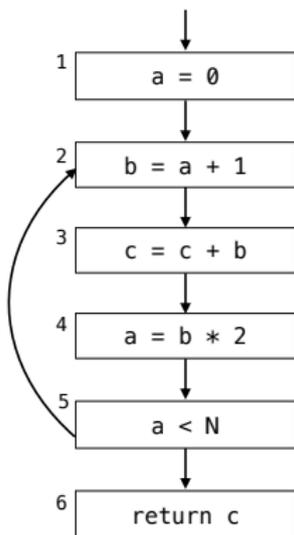
- A variable is *live* at program point p if its value could be used in the future (along some path starting at p).



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

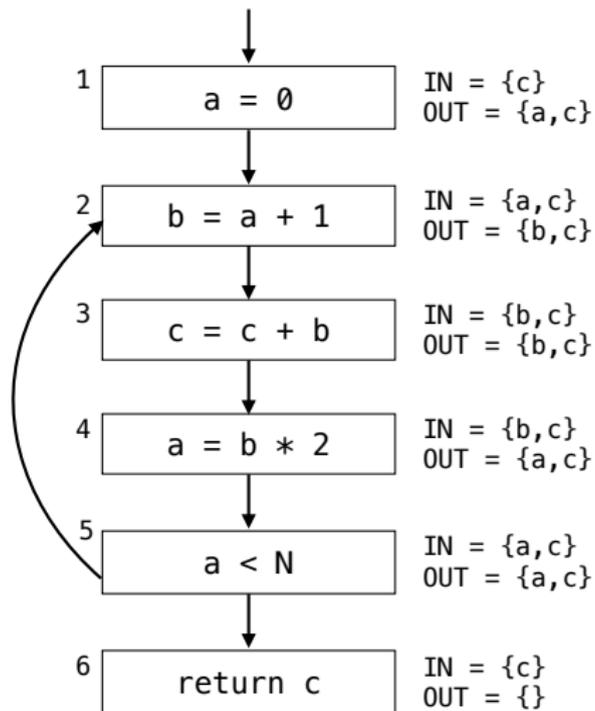
Example: Liveness of Variables

We analyze liveness from the future to the past.



- The live range of b : $\{2 \rightarrow 3, 3 \rightarrow 4\}$
- The live range of a : $\{1 \rightarrow 2, 4 \rightarrow 5 \rightarrow 2\}$ (not from $2 \rightarrow 3 \rightarrow 4$)
- 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: $n: x = y + z$
 - ▶ 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.

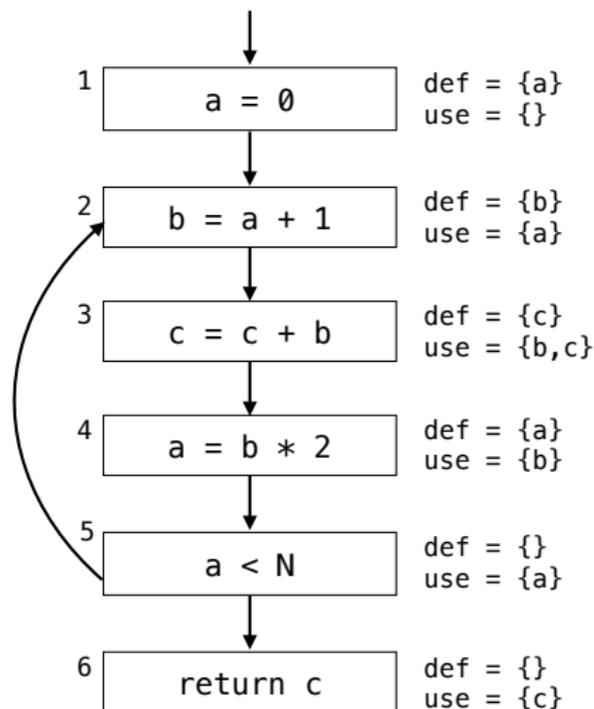
Liveness Analysis

The goal is to compute

$$\begin{aligned} \mathbf{in} & : \mathit{Block} \rightarrow 2^{\mathit{Var}} \\ \mathbf{out} & : \mathit{Block} \rightarrow 2^{\mathit{Var}} \end{aligned}$$

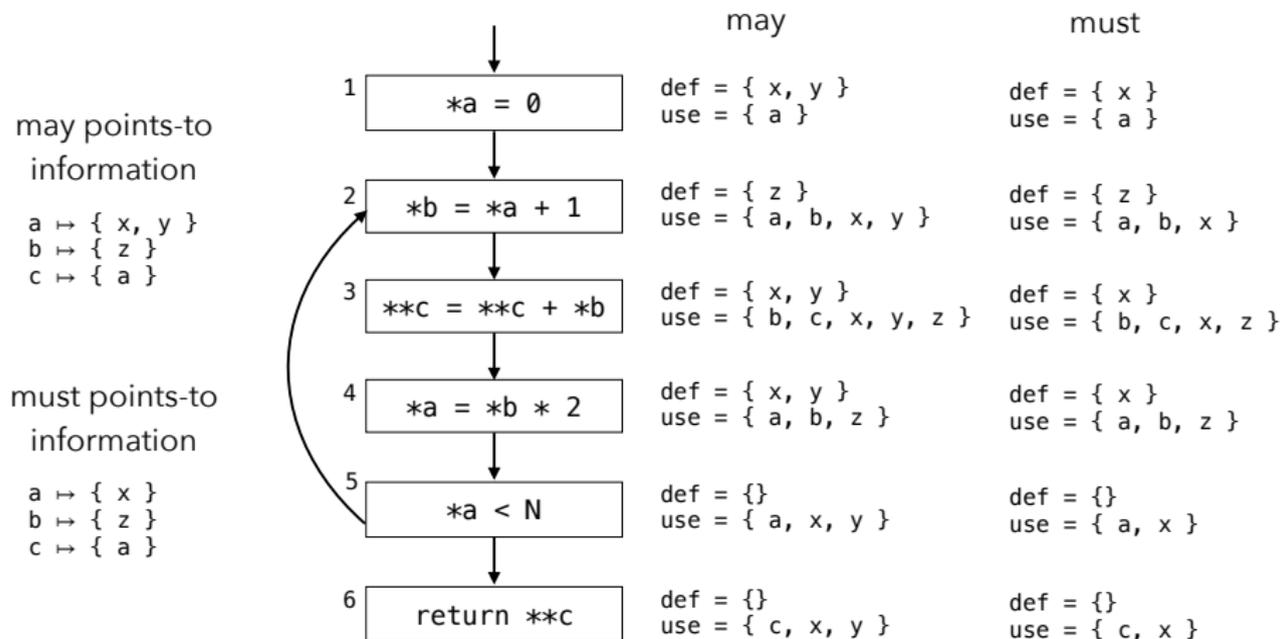
- 1 Compute def/use sets.
- 2 Derive transfer functions for each basic block in terms of def/use sets.
- 3 Derive the set of data-flow equations.
- 4 Solve the equation by the iterative fixed point algorithm.

Def/Use Sets



cf) Def/Use sets are only dynamically computable

In general, we need *pointer analysis* to compute (may/must) def/use sets.



Data-Flow Equations

Intuitions:

- 1 If a variable is in **use**(B), then it is live on entry to block B .
(Assume B has a single statement for simplicity)
- 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 entry to block B , then it is live at the end of predecessors of B .

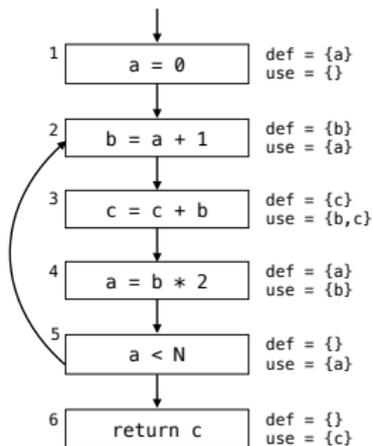
Equations:

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

Fixed Point Computation

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

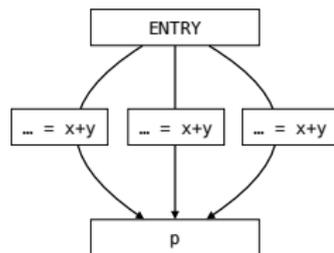
Example



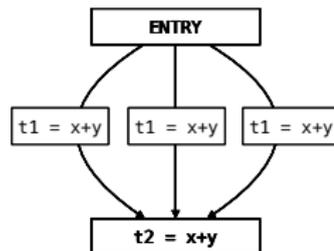
			1st		2nd		3rd	
	use	def	out	in	out	in	out	in
6	{c}	\emptyset	\emptyset	{c}	\emptyset	{c}	\emptyset	{c}
5	{a}	\emptyset	{c}	{a, c}	{a, c}	{a, c}	{a, c}	{a, c}
4	{b}	{a}	{a, c}	{b, c}	{a, c}	{b, c}	{a, c}	{b, c}
3	{b, c}	{c}	{b, c}	{b, c}	{b, c}	{b, c}	{b, c}	{b, c}
2	{a}	{b}	{b, c}	{a, c}	{b, c}	{a, c}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



- 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

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

- 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)
$x = y + z$	$\{y + z\} - \mathbf{kill}(s)$	expressions containing x
$x = \text{alloc}(n)$	\emptyset	expressions containing x
$x = y[i]$	$\{y[i]\} - \mathbf{kill}(s)$	expressions containing x
$x[i] = y$	\emptyset	expressions of the form $x[k]$

($x = y + z$ generates $y + z$, but $y = y + z$ does not because y is subsequently killed.)

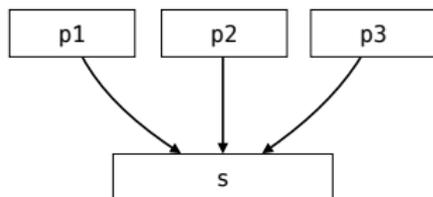
- What expressions are generated and killed by the block?

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

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:

$$\mathbf{in}(ENTRY) = \emptyset$$

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

$$\mathbf{in}(B) = \bigcap_{P \rightarrow B} \mathbf{out}(P)$$

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

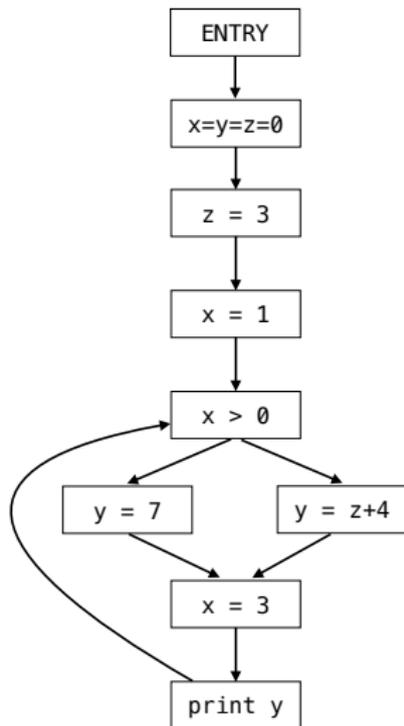
$$\mathbf{in}(B_i) = \bigcap_{P \hookrightarrow B_i} \mathbf{out}(P)$$

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

}

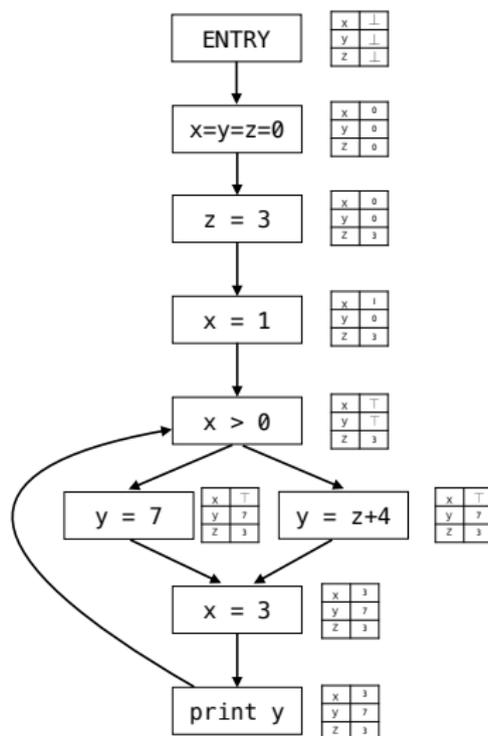
Constant Folding

Decide that the value of an expression is a constant and use it instead.

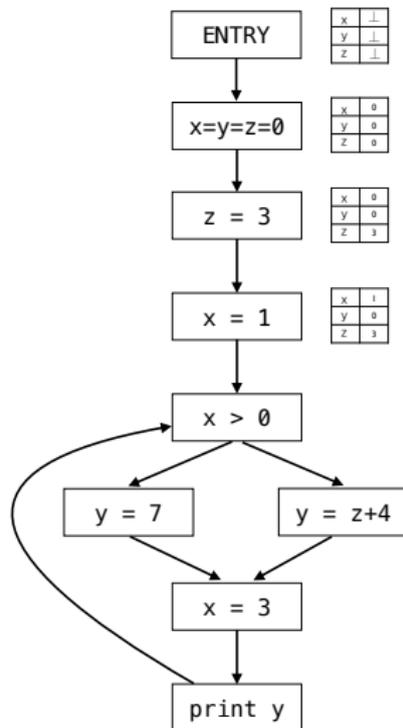


Constant Propagation Analysis

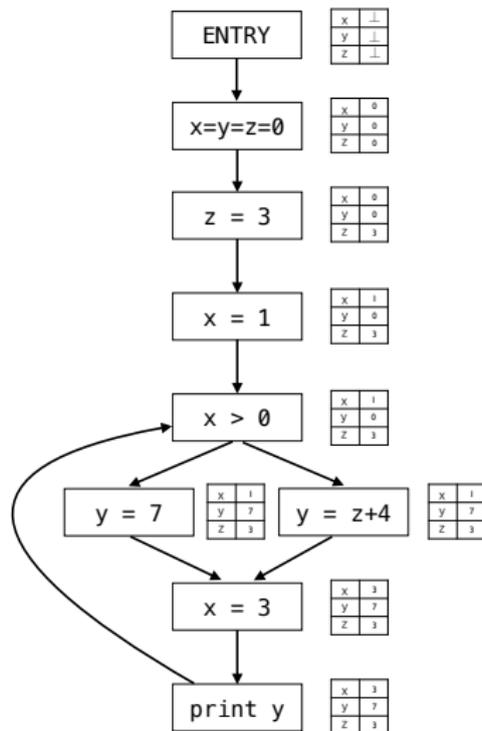
For each program point, determine whether a variable has a constant value whenever execution reaches that point.



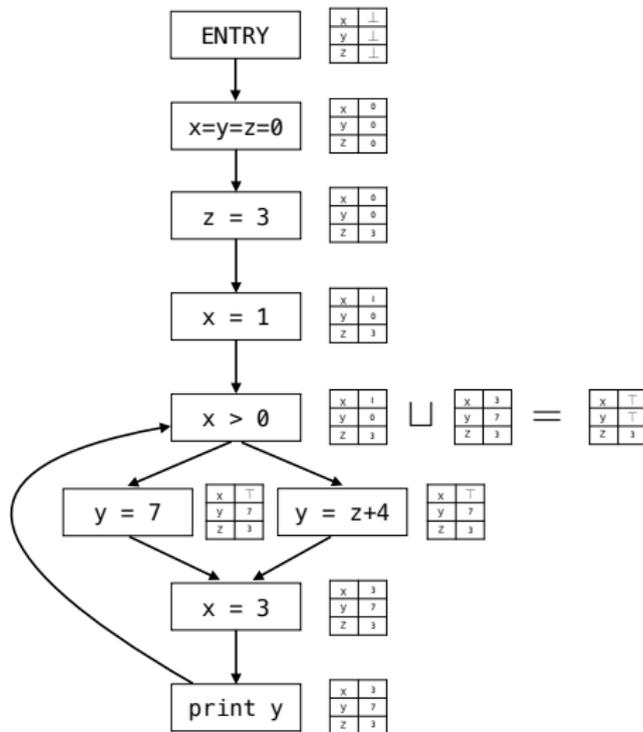
How It Works (1)



How It Works (2)



How It Works (3)

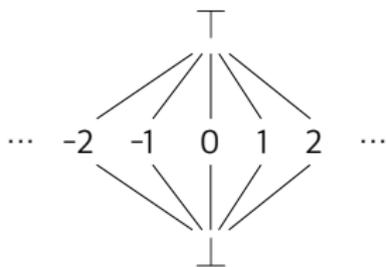


Constant Analysis

The goal is to compute

$$\begin{aligned} \mathbf{in} &: \mathit{Block} \rightarrow (\mathit{Var} \rightarrow \mathbb{C}) \\ \mathbf{out} &: \mathit{Block} \rightarrow (\mathit{Var} \rightarrow \mathbb{C}) \end{aligned}$$

where \mathbb{C} is a partially ordered set:



with the order:

$$\forall c_1, c_2 \in \mathbb{C}. c_1 \sqsubseteq c_2 \text{ iff } c_1 = \perp \vee c_2 = \top \vee c_1 = c_2$$

Functions in $\mathit{Var} \rightarrow \mathbb{C}$ are also partially ordered:

$$\forall d_1, d_2 \in (\mathit{Var} \rightarrow \mathbb{C}). d_1 \sqsubseteq d_2 \text{ iff } \forall x \in \mathit{Var}. d_1(x) \sqsubseteq d_2(x)$$

Join (Least Upper Bound)

The *join* between domain elements:

$$c_1 \sqcup c_2 = \begin{cases} c_2 & c_1 = \perp \\ c_1 & c_2 = \perp \\ c_1 & c_1 = c_2 \\ \top & \text{o.w.} \end{cases}$$

The join between abstract states:

$$d_1 \sqcup d_2 = \lambda x \in \text{Var. } d_1(x) \sqcup d_2(x)$$

Transfer Function

The transfer function

$$f_B : (\mathit{Var} \rightarrow \mathbb{C}) \rightarrow (\mathit{Var} \rightarrow \mathbb{C})$$

models the program execution in terms of the abstract values: e.g.,

- Transfer function for $z = 3$:

$$\lambda d. [z \mapsto 3]d$$

- Transfer function for $x > 0$:

$$\lambda d. d$$

- Transfer function for $y = z + 4$:

$$\lambda d. \begin{cases} [y \mapsto \perp]d & d(z) = \perp \\ [y \mapsto \top]d & d(z) = \top \\ [y \mapsto d(z) + 4]d & \text{o.w.} \end{cases}$$

Transfer Function

A simple set of commands:

$$\begin{aligned}c &\rightarrow x := e \mid x > n \mid \\e &\rightarrow n \mid x \mid e_1 + e_2 \mid e_1 - e_2\end{aligned}$$

The transfer function:

$$\begin{aligned}f_{x:=e}(d) &= [x \mapsto [e](d)]d \\f_{x>n}(d) &= d \\[n](d) &= n \\[x](d) &= d(x) \\[e_1 + e_2](d) &= [e_1](d) + [e_2](d) \\[e_1 - e_2](d) &= [e_1](d) - [e_2](d)\end{aligned}$$

Data-Flow Equations

Equation:

$$\mathbf{in}(B) = \bigsqcup_{P \hookrightarrow B} \mathbf{out}(P)$$
$$\mathbf{out}(B) = f_B(\mathbf{in}(B))$$

Fixed point computation:

For all i , $\mathbf{in}(B_i) = \mathbf{out}(B_i) = \lambda x. \perp$
while (changes to any **in** and **out** occur) {
 For all i , update
 $\mathbf{in}(B_i) = \bigsqcup_{P \hookrightarrow B_i} \mathbf{out}(P)$
 $\mathbf{out}(B_i) = f_{B_i}(\mathbf{in}(B_i))$
 }

Summary

- Data-flow analyses we covered:
 - ▶ Reaching definitions analysis
 - ▶ Liveness analysis
 - ▶ Available expressions analysis
 - ▶ Constant propagation analysis
- Optimization passes
 - ▶ Common subexpression elimination
 - ▶ Copy propagation
 - ▶ Decode elimination
 - ▶ Constant folding