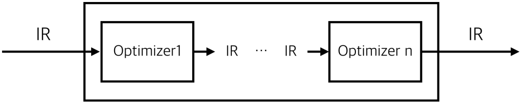
COSE312: Compilers

Lecture 14 — Optimization

Hakjoo Oh
2023 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               =>             u = x
y = u + 2                          y = u + 2
```

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.

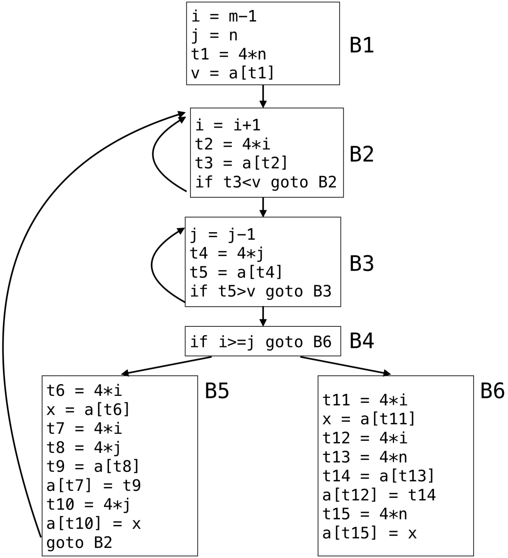
```
u = v     // deadcode
x = v + 1
u = x
y = u + 2
```

Constant Folding

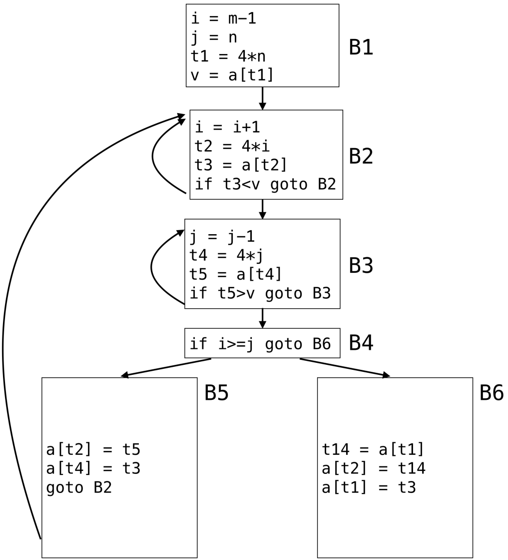
Decide that the value of an expression is a constant and use the constant instead.

```
c = 1                    c = 1
x = c + c      =>        x = 2
y = x + x                y = 4
```

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).

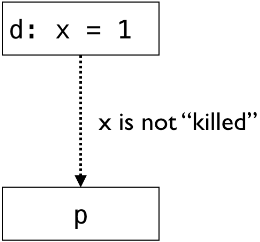
Data-Flow Analysis

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
- ...

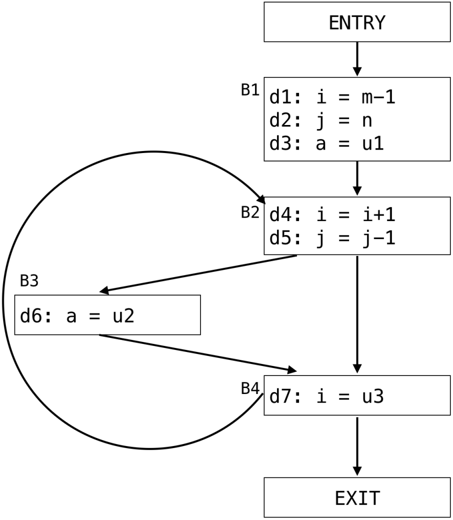
Reaching Definitions Analysis

- 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



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; ... }`
    ```

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

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

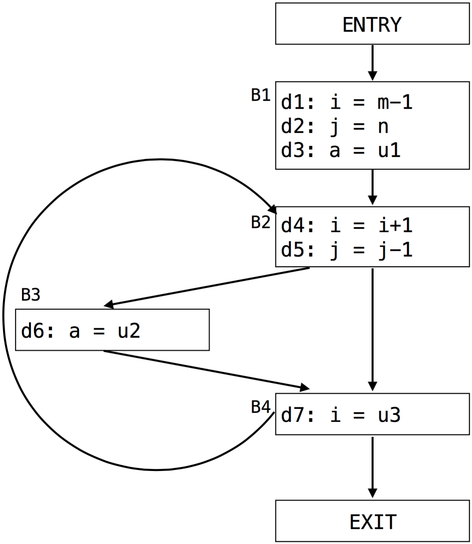
1. Compute gen/kill sets.
2. Derive transfer functions for each block in terms of gen/kill sets.
3. Derive the set of data-flow equations.
4. Solve the equation by the iterative fixed point algorithm.

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.

$B2$

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

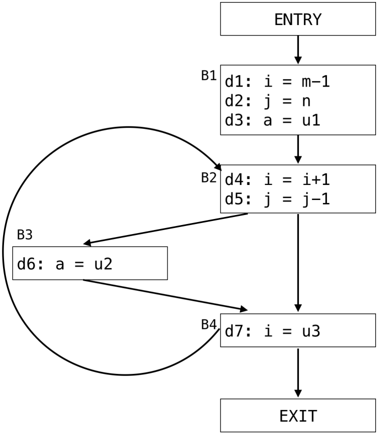
- The semantics of B is defined in terms of **gen**(B) and **kill**(B):

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

$B2$

```
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?):

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

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

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

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

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

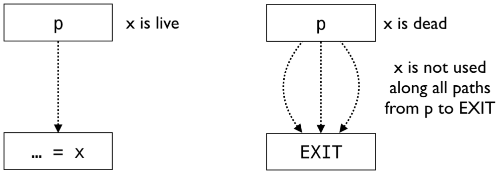
The Fixpoint Algorithm

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

Liveness Analysis

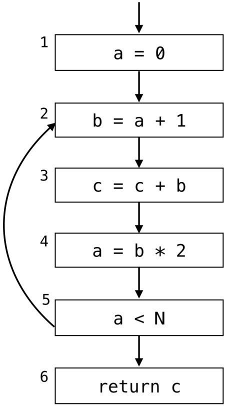
- A variable is *live* at program point p if its value could be used in the future (along some path starting at p).



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

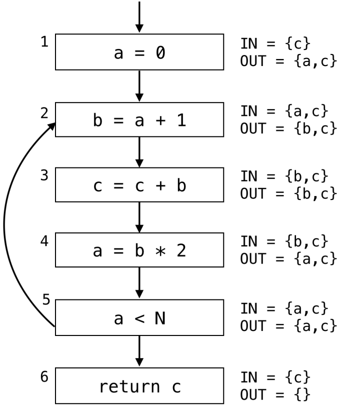
Example: Liveness of Variables

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.

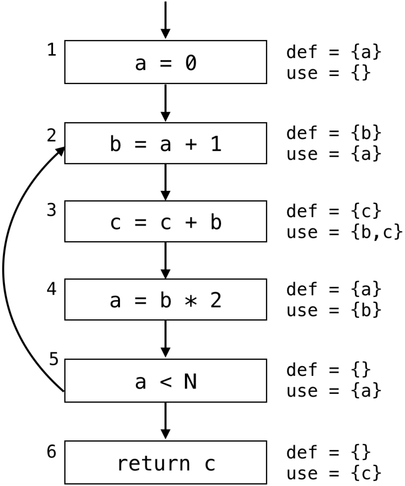
Liveness Analysis

The goal is to compute

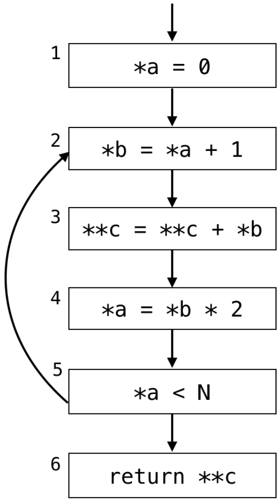
$$\begin{array}{rcl} \textbf{in} & : & Block \rightarrow 2^{Var} \\ \textbf{out} & : & Block \rightarrow 2^{Var} \end{array}$$

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.

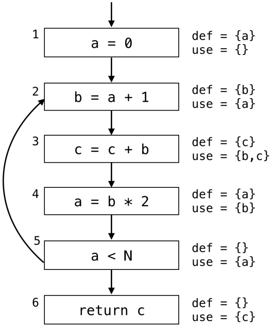
Equations:

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

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

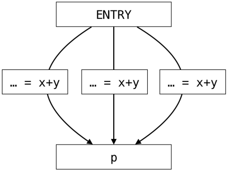
Example



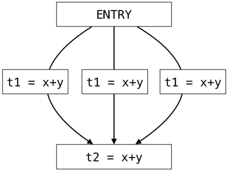
			$1st$		$2nd$		$3rd$	
	use	**def**	**out**	**in**	**out**	**in**	**out**	**in**
6	$\{c\}$	$\emptyset$	$\emptyset$	$\{c\}$	$\emptyset$	$\{c\}$	$\emptyset$	$\{c\}$
5	$\{a\}$	$\emptyset$	$\{c\}$	$\{a,c\}$	$\{a,c\}$	$\{a,c\}$	$\{a,c\}$	$\{a,c\}$
4	$\{b\}$	$\{a\}$	$\{a,c\}$	$\{b,c\}$	$\{a,c\}$	$\{b,c\}$	$\{a,c\}$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



- 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.

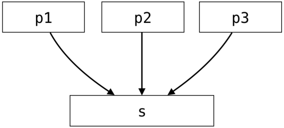
- What expressions are generated and killed by the block?

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

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:

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

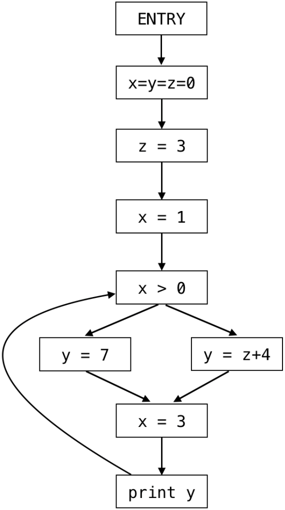
2. Solve the equations

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

$$
\begin{array}{l}
\textbf{in}(ENTRY) = \emptyset \\
\text{For other } B_i, \textbf{in}(B_i) = \textbf{out}(B_i) = Expr \\
\textbf{while} \text{ (changes to any } \textbf{in} \text{ and } \textbf{out} \text{ occur) } \{ \\
\quad \text{For all } i, \text{ update} \\
\qquad \textbf{in}(B_i) = \bigcap_{P \hookrightarrow B_i} \textbf{out}(P) \\
\qquad \textbf{out}(B_i) = \textbf{gen}(B_i) \cup (\textbf{in}(B_i) - \textbf{kill}(B_i)) \\
\}
\end{array}
$$

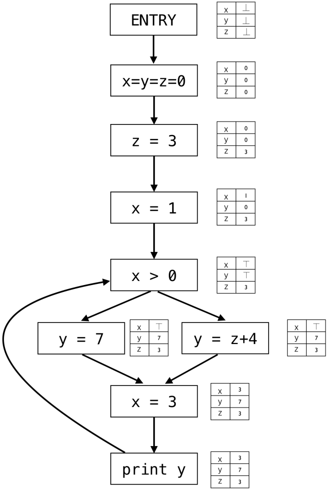
Constant Folding

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

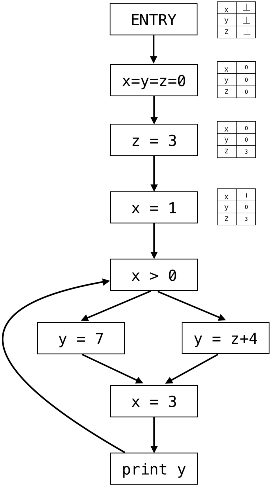


Constant Propagation Analysis

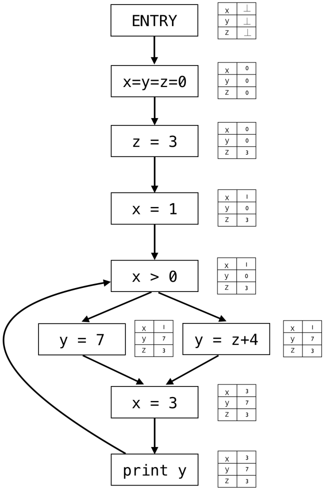
For each program point, determine whether a variable has a constant value whenever execution reaches that point.



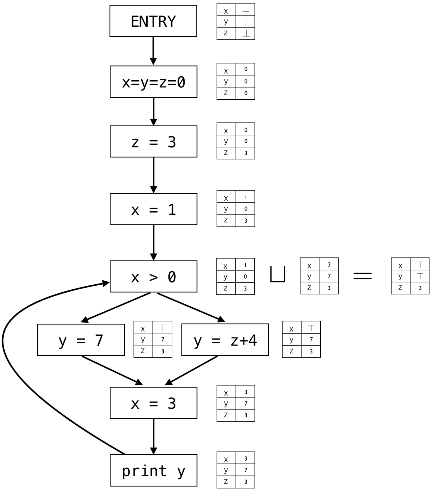
How It Works (1)



How It Works (2)



How It Works (3)

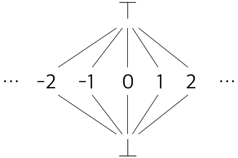


Constant Analysis

The goal is to compute

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

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



with the order:

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

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

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

Join (Least Upper Bound)

The *join* between domain elements:

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

The join between abstract states:

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

Transfer Function

The transfer function

$$f_B : (Var \to \mathbb{C}) \to (Var \to \mathbb{C})$$

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 $y = z + 4$:

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

Transfer Function

A simple set of commands:

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

The transfer function:

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

Data-Flow Equations

Equation:

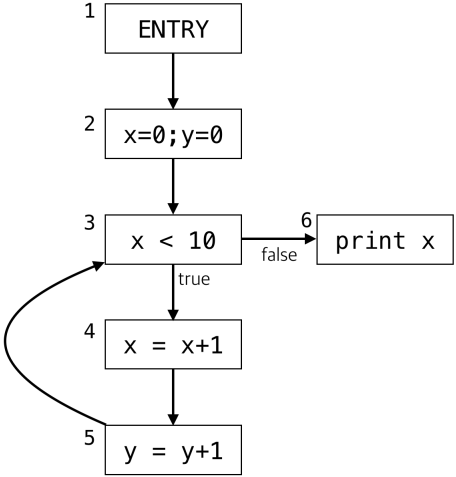
$$\mathbf{in}(B) = \bigsqcup_{P \hookrightarrow B} \mathbf{out}(P)$$

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

Fixed point computation:

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

Extension to Interval Analysis



Node	Result
1	$x \mapsto \bot$ $y \mapsto \bot$
2	$x \mapsto [0,0]$ $y \mapsto [0,0]$
3	$x \mapsto [0,9]$ $y \mapsto [0,+\infty]$
4	$x \mapsto [1,10]$ $y \mapsto [0,+\infty]$
5	$x \mapsto [1,10]$ $y \mapsto [1,+\infty]$
6	$x \mapsto [10,10]$ $y \mapsto [0,+\infty]$

Applications of Interval Analysis

- Static buffer-overflow detection: e.g.,

  ```
  ... a[x] ...
  ```

  where $\boldsymbol{a.size} = [\mathbf{10}, \mathbf{20}]$ and $\boldsymbol{x} = [\mathbf{5}, \mathbf{15}]$.
  - E.g., Sparrow[1] and Infer[2] use interval analysis to detect buffer overruns
- More precise constant analysis:

  ```
  if (...) {
    x = 1;  y = 2;
  } else {
    x = 2;  y = 1
  }
  z = x + y;
  ```
- Many others

[1] `http://www.fasoo.com/`

[2] `http://fbinfer.com`

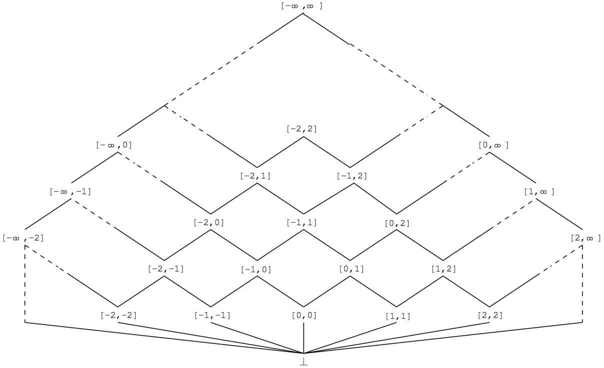
Interval Analysis

The goal is to compute

$$\begin{array}{rcl} \textbf{in} & : & \mathit{Block} \to (\mathit{Var} \to \mathbb{I}) \\ \textbf{out} & : & \mathit{Block} \to (\mathit{Var} \to \mathbb{I}) \end{array}$$

where $\mathbb{I}$ is a partially ordered set:

$$\mathbb{I} = \{\bot\} \cup \{[l, u] \mid l, u \in \mathbb{Z} \cup \{-\infty, +\infty\} \ \wedge \ l \leq u\}$$



Join (Least Upper Bound)

- The join operator merges multiple data flows: e.g.,
  - $[1, 3] \sqcup [2, 4] = [1, 4]$
  - $[1, 3] \sqcup [7, 9] = [1, 9]$
- Definition:

$$\begin{array}{rcl} \bot \sqcup i & = & i \\ i \sqcup \bot & = & i \\ [l_1, u_1] \sqcup [l_2, u_2] & = & [\min(l_1, l_2), \max(u_1, u_2)] \end{array}$$

- The join between abstract states:

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

Transfer Function

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

The transfer function:

$$\begin{array}{rcl} f_{x:=e}(d) & = & [x \mapsto [\![e]\!](d)]d \\ f_{x>n}(d) & = & [x \mapsto d(x) \sqcap [n+1, +\infty]]d \\ [\![n]\!](d) & = & [n, n] \\ [\![x]\!](d) & = & d(x) \\ [\![e_1 + e_2]\!](d) & = & [\![e_1]\!](d) \hat{+} [\![e_2]\!](d) \\ [\![e_1 - e_2]\!](d) & = & [\![e_1]\!](d) \hat{-} [\![e_2]\!](d) \end{array}$$

Data-Flow Equations

Equation:

$$\begin{aligned} \mathbf{in}(B) &= \bigsqcup_{P \hookrightarrow B} \mathbf{out}(P) \\ \mathbf{out}(B) &= f_B(\mathbf{in}(B)) \end{aligned}$$

Fixed point computation:

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

Fixed Point Computation Does Not Terminate

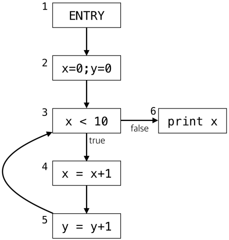
The conventional fixed point computation requires an infinite number of iterations to converge:

Node	initial	1	2	3	10	11	k	∞
1	$x \mapsto \bot$ $y \mapsto \bot$	$x \mapsto \bot$ $y \mapsto \bot$	$x \mapsto \bot$ $y \mapsto \bot$	$x \mapsto \bot$ $y \mapsto \bot$	$x \mapsto \bot$ $y \mapsto \bot$	$x \mapsto \bot$ $y \mapsto \bot$	$x \mapsto \bot$ $y \mapsto \bot$	$x \mapsto \bot$ $y \mapsto \bot$
2	$x \mapsto \bot$ $y \mapsto \bot$	$x \mapsto [0,0]$ $y \mapsto [0,0]$	$x \mapsto [0,0]$ $y \mapsto [0,0]$	$x \mapsto [0,0]$ $y \mapsto [0,0]$	$x \mapsto [0,0]$ $y \mapsto [0,0]$	$x \mapsto [0,0]$ $y \mapsto [0,0]$	$x \mapsto [0,0]$ $y \mapsto [0,0]$	$x \mapsto [0,0]$ $y \mapsto [0,0]$
3	$x \mapsto \bot$ $y \mapsto \bot$	$x \mapsto [0,0]$ $y \mapsto [0,0]$	$x \mapsto [0,1]$ $y \mapsto [0,1]$	$x \mapsto [0,2]$ $y \mapsto [0,2]$	$x \mapsto [0,9]$ $y \mapsto [0,9]$	$x \mapsto [0,9]$ $y \mapsto [0,10]$	$x \mapsto [0,9]$ $y \mapsto [0,k-1]$	$x \mapsto [0,9]$ $y \mapsto [0,+\infty]$
4	$x \mapsto \bot$ $y \mapsto \bot$	$x \mapsto [1,1]$ $y \mapsto [0,0]$	$x \mapsto [1,2]$ $y \mapsto [0,1]$	$x \mapsto [1,3]$ $y \mapsto [0,2]$	$x \mapsto [1,10]$ $y \mapsto [0,9]$	$x \mapsto [1,10]$ $y \mapsto [0,10]$	$x \mapsto [1,10]$ $y \mapsto [0,k-1]$	$x \mapsto [1,10]$ $y \mapsto [0,+\infty]$
5	$x \mapsto \bot$ $y \mapsto \bot$	$x \mapsto [1,1]$ $y \mapsto [1,1]$	$x \mapsto [1,2]$ $y \mapsto [1,2]$	$x \mapsto [1,3]$ $y \mapsto [1,3]$	$x \mapsto [1,10]$ $y \mapsto [1,10]$	$x \mapsto [1,10]$ $y \mapsto [1,11]$	$x \mapsto [1,10]$ $y \mapsto [1,k]$	$x \mapsto [1,10]$ $y \mapsto [1,+\infty]$
6	$x \mapsto \bot$ $y \mapsto \bot$	$x \mapsto \bot$ $y \mapsto [0,0]$	$x \mapsto \bot$ $y \mapsto [0,1]$	$x \mapsto \bot$ $y \mapsto [0,2]$	$x \mapsto [10,10]$ $y \mapsto [0,9]$	$x \mapsto [10,10]$ $y \mapsto [0,10]$	$x \mapsto [10,10]$ $y \mapsto [0,k-1]$	$x \mapsto [10,10]$ $y \mapsto [0,+\infty]$

To ensure termination and precision, two staged fixed point computation is required:

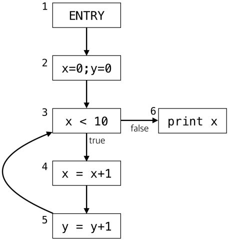
1. increasing widening sequence
2. decreasing narrowing sequence

1. Fixed Point Computation with Widening



Node	initial	1	2	3
1	$x \mapsto \bot$ $y \mapsto \bot$	$x \mapsto \bot$ $y \mapsto \bot$	$x \mapsto \bot$ $y \mapsto \bot$	$x \mapsto \bot$ $y \mapsto \bot$
2	$x \mapsto \bot$ $y \mapsto \bot$	$x \mapsto [0,0]$ $y \mapsto [0,0]$	$x \mapsto [0,0]$ $y \mapsto [0,0]$	$x \mapsto [0,0]$ $y \mapsto [0,0]$
3	$x \mapsto \bot$ $y \mapsto \bot$	$x \mapsto [0,0]$ $y \mapsto [0,0]$	$x \mapsto [0,9]$ $y \mapsto [0,+\infty]$	$x \mapsto [0,9]$ $y \mapsto [0,+\infty]$
4	$x \mapsto \bot$ $y \mapsto \bot$	$x \mapsto [1,1]$ $y \mapsto [0,0]$	$x \mapsto [1,10]$ $y \mapsto [0,+\infty]$	$x \mapsto [1,10]$ $y \mapsto [0,+\infty]$
5	$x \mapsto \bot$ $y \mapsto \bot$	$x \mapsto [1,1]$ $y \mapsto [1,1]$	$x \mapsto [1,10]$ $y \mapsto [1,+\infty]$	$x \mapsto [1,10]$ $y \mapsto [1,+\infty]$
6	$x \mapsto \bot$ $y \mapsto \bot$	$x \mapsto \bot$ $y \mapsto [0,0]$	$x \mapsto [10,+\infty]$ $y \mapsto [0,+\infty]$	$x \mapsto [10,+\infty]$ $y \mapsto [0,+\infty]$

2. Fixed Point Computation with Narrowing



Node	initial	1	2
1	$x \mapsto \bot$ $y \mapsto \bot$	$x \mapsto \bot$ $y \mapsto \bot$	$x \mapsto \bot$ $y \mapsto \bot$
2	$x \mapsto [0,0]$ $y \mapsto [0,0]$	$x \mapsto [0,0]$ $y \mapsto [0,0]$	$x \mapsto [0,0]$ $y \mapsto [0,0]$
3	$x \mapsto [0,9]$ $y \mapsto [0,+\infty]$	$x \mapsto [0,9]$ $y \mapsto [0,+\infty]$	$x \mapsto [0,9]$ $y \mapsto [0,+\infty]$
4	$x \mapsto [1,10]$ $y \mapsto [0,+\infty]$	$x \mapsto [1,10]$ $y \mapsto [0,+\infty]$	$x \mapsto [1,10]$ $y \mapsto [0,+\infty]$
5	$x \mapsto [1,10]$ $y \mapsto [1,+\infty]$	$x \mapsto [1,10]$ $y \mapsto [1,+\infty]$	$x \mapsto [1,10]$ $y \mapsto [1,+\infty]$
6	$x \mapsto [10,+\infty]$ $y \mapsto [0,+\infty]$	$x \mapsto [10,10]$ $y \mapsto [0,+\infty]$	$x \mapsto [10,10]$ $y \mapsto [0,+\infty]$

Simple Widening and Narrowing Operators

A simple widening operator for the Interval domain:

$$\begin{array}{rcll} [a,b] & \triangledown & \bot & = [a,b] \\ \bot & \triangledown & [c,d] & = [c,d] \\ [a,b] & \triangledown & [c,d] & = [(c < a ? -\infty : a), (b < d ? +\infty : b)] \end{array}$$

A simple narrowing operator:

$$\begin{array}{rcll} [a,b] & \triangle & \bot & = \bot \\ \bot & \triangle & [c,d] & = \bot \\ [a,b] & \triangle & [c,d] & = [(a = -\infty ? c : a), (b = +\infty ? d : b)] \end{array}$$

Widening and narrowing for abstract states:

$$d_1 \triangledown d_2 = \lambda x \in Var.\ d_1(x) \triangledown d_2(x)$$

$$d_1 \triangle d_2 = \lambda x \in Var.\ d_1(x) \triangle d_2(x)$$

Fixed Point Computation with Widening/Narrowing

For all $i, \textbf{in}(B_i) = \textbf{out}(B_i) = \lambda x.\bot$
while (changes to any **in** and **out** occur) {
  For all i, update
    $\textbf{in}(B_i) = \textbf{in}(B_i) \bigtriangledown \left(\bigsqcup_{P \hookrightarrow B_i} \textbf{out}(P)\right)$
    $\textbf{out}(B_i) = \textbf{out}(B_i) \bigtriangledown f_{B_i}(\textbf{in}(B_i))$
}
while (changes to any **in** and **out** occur) {
  For all i, update
    $\textbf{in}(B_i) = \textbf{in}(B_i) \bigtriangleup \left(\bigsqcup_{P \hookrightarrow B_i} \textbf{out}(P)\right)$
    $\textbf{out}(B_i) = \textbf{out}(B_i) \bigtriangleup f_{B_i}(\textbf{in}(B_i))$
}

Summary

- Data-flow analyses we covered:
  - Reaching definitions analysis
  - Liveness analysis
  - Available expressions analysis
  - Constant propagation analysis
  - Interval analysis
- Foundational theory: "Abstract Interpretation"