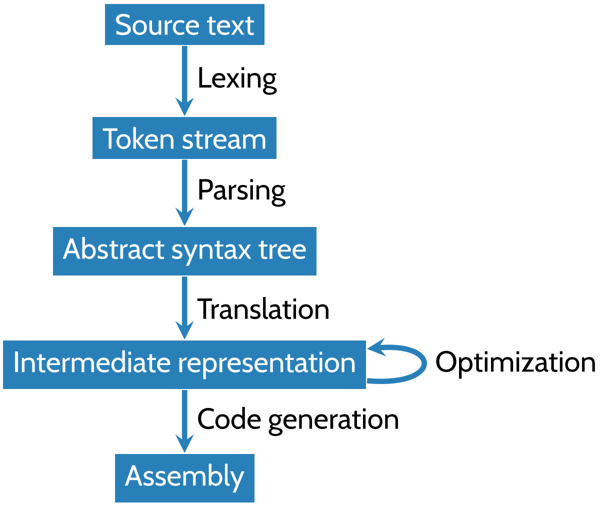
COS320: Compiling Techniques

Zak Kincaid

March 26, 2024

Analysis and Optimization

Compiler phases (simplified)



Optimization

- Optimization operates as a sequence of IR-to-IR transformations. Each transformation is expected to:
  - *improve performance* (time, space, power)
  - *not change the high-level (defined) behavior of the program*
- Each optimization pass does something small and simple.
  - *Combination* of passes can yield sophisticated transformations

Optimization

- Optimization operates as a sequence of IR-to-IR transformations. Each transformation is expected to:
  - *improve performance* (time, space, power)
  - *not change the high-level (defined) behavior of the program*
- Each optimization pass does something small and simple.
  - *Combination* of passes can yield sophisticated transformations
- Optimization simplifies compiler writing
  - More modular: can translate to IR in a simple-but-inefficient way, then optimize
- Optimization simplifies programming
  - Programmer can spend less time thinking about low-level performance issues
  - More portable: compiler can take advantage of the characteristics of a particular machine

Algebraic simplification

Idea: replace complex expressions with simpler / cheaper ones

$$e * 1 \rightarrow e$$

$$0 + e \rightarrow e$$

$$2 * 3 \rightarrow 6$$

$$-(-e) \rightarrow e$$

$$e * 4 \rightarrow e \text{«} 2$$

$$\ldots$$

Loop unrolling

- Idea: avoid branching by trading space for time.
- Can expose opportunities for using SIMD instructions

```
long array_sum (long *a, long n) {
  long i;
  long sum = 0;
  for (i = 0; i < n; i++) {
    sum += *(a + i);
  }
  return sum;
}
```

→

```
long array_sum (long *a, long n) {
  long i;
  long sum = 0;
  for (i = 0; i < n % 4; i++) {
    sum += *(a + i);
  }
  for (; i < n; i += 4) {
    sum += *(a + i);
    sum += *(a + i + 1);
    sum += *(a + i + 2);
    sum += *(a + i + 3);
  }
  return sum;
}
```

Strength reduction

Idea: replace expensive operation (e.g., multiplication) w/ cheaper one (e.g., addition).

```
long trace (long *m, long n) {
  long i;
  long result = 0;
  for (i = 0; i < n; i++) {
    result += *(m + i*n + i);
  }
  return result;
}
```

→

```
long trace (long *m, long n) {
  long i;
  long result = 0;
  long *next = m;
  for (i = 0; i < n; i++) {
    result += *next;
    next += n + 1;
  }
  return result;
}
```

Optimization and Analysis

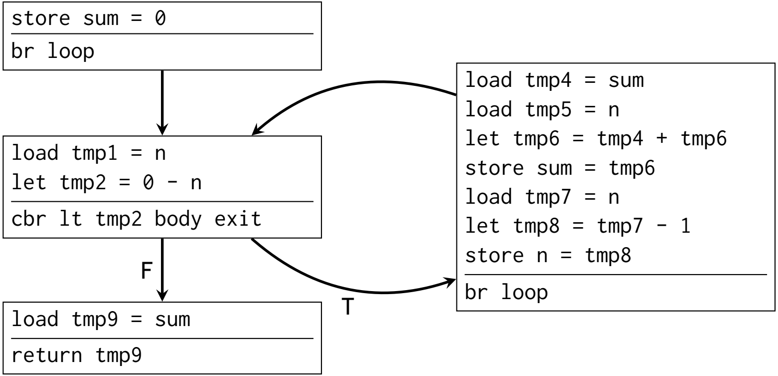
- *Program analysis*: conservatively approximate the run-time behavior of a program at compile time.
  - Type inference: find the type of value each expression will evaluate to at run time. *Conservative* in the sense that the analysis will abort if it cannot find a type for a variable, even if one exists.
  - Constant propagation: if a variable only holds on value at run time, find that value. *Conservative* in the sense that analysis may fail to find constant values for variables that have them.

Optimization and Analysis

- *Program analysis*: conservatively approximate the run-time behavior of a program at compile time.
  - Type inference: find the type of value each expression will evaluate to at run time. *Conservative* in the sense that the analysis will abort if it cannot find a type for a variable, even if one exists.
  - Constant propagation: if a variable only holds on value at run time, find that value. *Conservative* in the sense that analysis may fail to find constant values for variables that have them.
- Optimization passes are typically informed by analysis
  - Analysis lets us know which transformations are safe
  - Conservative analysis $\Rightarrow$ never perform an unsafe optimization, but may miss some safe optimizations.

Control Flow Graphs (CFG)

```
int sum_upto(int n) {
  int sum = 0;
  while (n > 0) {
    sum += n;
    n--;
  }
  return sum;
}
```



- Control flow graphs are one of the basic data structures used to represent programs in many program analyses
- Recall: A *control flow graph* (CFG) for a procedure P is a directed, rooted graph $G = (N, E, r)$ where
    - The nodes are basic blocks of P
    - There is an edge $n_i \to n_j \in E$ iff n_j may execute immediately after n_i
    - There is a distinguished entry block r where the execution of the procedure begins

Simple imperative language

- Suppose that we have the following language:

```
  <instr> ::= <var> = add <opn>, <opn>
            | <var> = mul <opn>, <opn>
            | <var> = opn
    <opn> ::= <int> | <var>
  <block> ::= <instr><block> | <term>
   <term> ::= blez <opn>, <label>, <label>
            | return <opn>
<program> ::= <program> <label> : <block> | <block>
```

- Note: no uids, no SSA
  - We'll take a look at how SSA affects program analysis later

Constant propagation

- The goal of constant propagation: determine at each instruction I a *constant environment*
  - A constant environment is a symbol table mapping each variable x to one of:
    - an integer n (indicating that x's value is n whenever the program is at I)
    - $\top$ (indicating that x might take more than one value at I)
    - $\bot$ (indicating that x may take no values at run-time – I is unreachable)
- Motivation: can evaluate expressions at compile time to save on run time

```
x = add 1, 2
y = mul x, 11
z = add x, y
```

Constant propagation

- The goal of constant propagation: determine at each instruction I a *constant environment*
  - A constant environment is a symbol table mapping each variable x to one of:
    - an integer n (indicating that x's value is n whenever the program is at I)
    - $\top$ (indicating that x might take more than one value at I)
    - $\bot$ (indicating that x may take no values at run-time – I is unreachable)
- Motivation: can evaluate expressions at compile time to save on run time

$$\{x \mapsto \top, y \mapsto \top, z \mapsto \top\}$$

```
x = add 1, 2
y = mul x, 11
z = add x, y
```

Constant propagation

- The goal of constant propagation: determine at each instruction I a *constant environment*
  - A constant environment is a symbol table mapping each variable x to one of:
    - an integer n (indicating that x's value is n whenever the program is at I)
    - $\top$ (indicating that x might take more than one value at I)
    - $\bot$ (indicating that x may take no values at run-time – I is unreachable)
- Motivation: can evaluate expressions at compile time to save on run time

$\{\mathsf{x} \mapsto \top, \mathsf{y} \mapsto \top, \mathsf{z} \mapsto \top\}$

$\{\mathsf{x} \mapsto 3, \mathsf{y} \mapsto \top, \mathsf{z} \mapsto \top\}$

```
x = add 1, 2
y = mul x, 11
z = add x, y
```

Constant propagation

- The goal of constant propagation: determine at each instruction I a *constant environment*
  - A constant environment is a symbol table mapping each variable x to one of:
    - an integer n (indicating that x's value is n whenever the program is at I)
    - $\top$ (indicating that x might take more than one value at I)
    - $\bot$ (indicating that x may take no values at run-time – I is unreachable)
- Motivation: can evaluate expressions at compile time to save on run time

$\{\mathsf{x} \mapsto \top, \mathsf{y} \mapsto \top, \mathsf{z} \mapsto \top\}$

$\{\mathsf{x} \mapsto 3, \mathsf{y} \mapsto \top, \mathsf{z} \mapsto \top\}$

$\{\mathsf{x} \mapsto 3, \mathsf{y} \mapsto 33, \mathsf{z} \mapsto \top\}$

```
x = add 1, 2
y = mul x, 11
z = add x, y
```

Constant propagation

- The goal of constant propagation: determine at each instruction I a *constant environment*
  - A constant environment is a symbol table mapping each variable x to one of:
    - an integer n (indicating that x's value is n whenever the program is at I)
    - $\top$ (indicating that x might take more than one value at I)
    - $\bot$ (indicating that x may take no values at run-time – I is unreachable)
- Motivation: can evaluate expressions at compile time to save on run time

$$\{\mathsf{x} \mapsto \top, \mathsf{y} \mapsto \top, \mathsf{z} \mapsto \top\}$$

$$\{\mathsf{x} \mapsto 3, \mathsf{y} \mapsto \top, \mathsf{z} \mapsto \top\}$$

$$\{\mathsf{x} \mapsto 3, \mathsf{y} \mapsto 33, \mathsf{z} \mapsto \top\}$$

```
x = 3
y = mul x, 11
z = add x, y
```

Constant propagation

- The goal of constant propagation: determine at each instruction I a *constant environment*
  - A constant environment is a symbol table mapping each variable x to one of:
    - an integer n (indicating that x's value is n whenever the program is at I)
    - $\top$ (indicating that x might take more than one value at I)
    - $\bot$ (indicating that x may take no values at run-time – I is unreachable)
- Motivation: can evaluate expressions at compile time to save on run time

$\{x \mapsto \top, y \mapsto \top, z \mapsto \top\}$

```
x = 3
```

$\{x \mapsto 3, y \mapsto \top, z \mapsto \top\}$

```
y = 33
```

$\{x \mapsto 3, y \mapsto 33, z \mapsto \top\}$

```
z = add x, y
```

Constant propagation

- The goal of constant propagation: determine at each instruction I a *constant environment*
  - A constant environment is a symbol table mapping each variable x to one of:
    - an integer n (indicating that x's value is n whenever the program is at I)
    - $\top$ (indicating that x might take more than one value at I)
    - $\bot$ (indicating that x may take no values at run-time – I is unreachable)
- Motivation: can evaluate expressions at compile time to save on run time

$\{\mathsf{x} \mapsto \top, \mathsf{y} \mapsto \top, \mathsf{z} \mapsto \top\}$

$\{\mathsf{x} \mapsto 3, \mathsf{y} \mapsto \top, \mathsf{z} \mapsto \top\}$

$\{\mathsf{x} \mapsto 3, \mathsf{y} \mapsto 33, \mathsf{z} \mapsto \top\}$

```
x = 3
y = 33
z = 36
```

Propagating constants through instructions

- Goal: given a constant environment C and an instruction
  - $x = \texttt{add}\ opn_1, opn_2$
  - $x = \texttt{mul}\ opn_1, opn_2$
  - $x = opn$

  Assuming that constant environment C holds *before* the instruction, what is the constant environment *after* the instruction?

Propagating constants through instructions

- Goal: given a constant environment C and an instruction
  - $x = \texttt{add}\ opn_1, opn_2$
  - $x = \texttt{mul}\ opn_1, opn_2$
  - $x = opn$

  Assuming that constant environment C holds *before* the instruction, what is the constant environment *after* the instruction?
- Define an evaluator for operands:

$$\mathit{eval}(\mathit{opn}, C) = \begin{cases} C(\mathit{opn}) & \text{if opn is a variable} \\ \mathit{opn} & \text{if opn is an int} \end{cases}$$

Propagating constants through instructions

- Goal: given a constant environment C and an instruction
  - $x = \texttt{add}\ opn_1, opn_2$
  - $x = \texttt{mul}\ opn_1, opn_2$
  - $x = opn$

  Assuming that constant environment C holds *before* the instruction, what is the constant environment *after* the instruction?
- Define an evaluator for operands:

$$\textit{eval}(\textit{opn}, C) = \begin{cases} C(\textit{opn}) & \text{if opn is a variable} \\ \textit{opn} & \text{if opn is an int} \end{cases}$$

- Define an evaluator for instructions:

$$post(instr, C) = \begin{cases} \bot & \text{if } C \text{ is } \bot \\ C\{x \mapsto eval(opn, C)\} & \text{if instr is } x = opn \\ C\{x \mapsto \top\} & \text{if } eval(opn_1, C) = \top \vee eval(opn_2, C) = \top \\ C\{x \mapsto eval(opn_1, C) + eval(opn_2, C)\} & \text{if instr is } x = \texttt{add}\ \textit{opn}_1, \textit{opn}_2 \\ C\{x \mapsto eval(opn_1, C) * eval(opn_2, C)\} & \text{if instr is } x = \texttt{mul}\ \textit{opn}_1, \textit{opn}_2 \end{cases}$$

Propagating constants through basic blocks

- How do we propagate a constant environment through a basic block?

Propagating constants through basic blocks

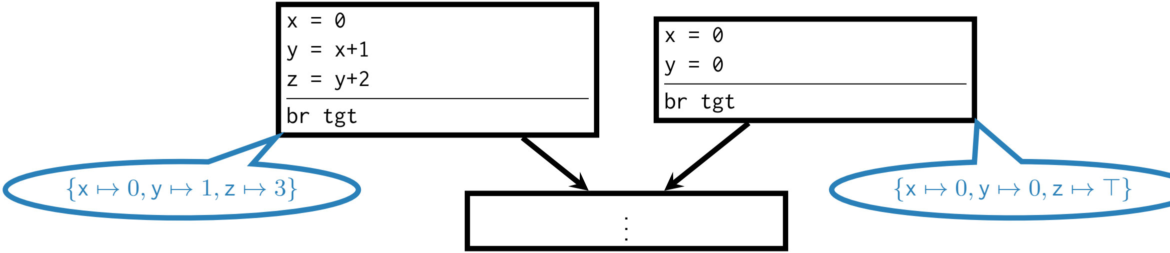
- How do we propagate a constant environment through a basic block?
- Block takes the form $instr_1, \ldots, instr_n, term$.
  take $post(block, C) = post(instr_n, \ldots post(instr_1, C) \ldots)$

Propagating constants across edges

- If a block has exactly one predecessor: constant environment at entry is constant environment at exit of predecessor

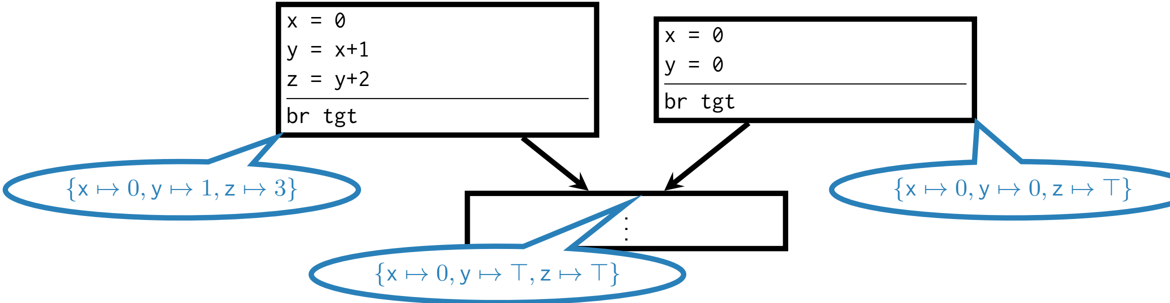
Propagating constants across edges

- If a block has exactly one predecessor: constant environment at entry is constant environment at exit of predecessor
- If a block has multiple predecessors, must combine constant environments of both:



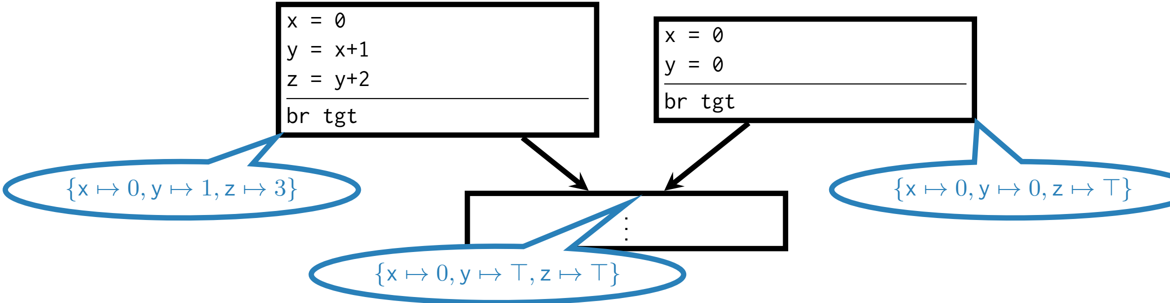
Propagating constants across edges

- If a block has exactly one predecessor: constant environment at entry is constant environment at exit of predecessor
- If a block has multiple predecessors, must combine constant environments of both:



Propagating constants across edges

- If a block has exactly one predecessor: constant environment at entry is constant environment at exit of predecessor
- If a block has multiple predecessors, must combine constant environments of both:
- Merge operator $\sqcup$ defined as:
  - $e \sqcup \bot = \bot \sqcup e = e$
  - $(e_1 \sqcup e_2)(x) = \begin{cases} e_1(x) & \text{if } e_1(x) = e_2(x) \\ \top & \text{otherwise} \end{cases}$



Propagating constants through control flow graphs

- For *acyclic graphs*:

Propagating constants through control flow graphs

- For *acyclic graphs*: topologically sort basic blocks, propagate constant environments forward
    - Constant environment for entry node maps each variable to $\top$

Propagating constants through control flow graphs

- For *acyclic graphs*: topologically sort basic blocks, propagate constant environments forward
  - Constant environment for entry node maps each variable to $\top$
- What about loops?

- Recall: a partial order $\sqsubseteq$ is a binary relation that is
  - Reflexive: $a \sqsubseteq a$
  - Transitive: $a \sqsubseteq b$ and $b \sqsubseteq c$ implies $a \sqsubseteq c$
  - Antisymmetric: $a \sqsubseteq b$ and $b \sqsubseteq a$ implies $a = b$
- Examples: the subset relation, the divisibility relation on the naturals, ...

- Recall: a partial order $\sqsubseteq$ is a binary relation that is
  - Reflexive: $a \sqsubseteq a$
  - Transitive: $a \sqsubseteq b$ and $b \sqsubseteq c$ implies $a \sqsubseteq c$
  - Antisymmetric: $a \sqsubseteq b$ and $b \sqsubseteq a$ implies $a = b$
- Examples: the subset relation, the divisibility relation on the naturals, ...
- Place a partial order on $\mathbb{Z} \cup \{\bot, \top\}$: $\bot \sqsubseteq n \sqsubseteq \top$ (most information to least information)

- Recall: a partial order $\sqsubseteq$ is a binary relation that is
  - Reflexive: $a \sqsubseteq a$
  - Transitive: $a \sqsubseteq b$ and $b \sqsubseteq c$ implies $a \sqsubseteq c$
  - Antisymmetric: $a \sqsubseteq b$ and $b \sqsubseteq a$ implies $a = b$
- Examples: the subset relation, the divisibility relation on the naturals, ...
- Place a partial order on $\mathbb{Z} \cup \{\bot, \top\}$: $\bot \sqsubseteq n \sqsubseteq \top$ (most information to least information)
- Lift the ordering to constant environments: $f \sqsubseteq g$ iff $f(x) \sqsubseteq g(x)$ for all x
  - $f \sqsubseteq g$: f is a "better" constant environment than g
  - f sends x to $\top$ implies g sends x to $\top$

- Recall: a partial order $\sqsubseteq$ is a binary relation that is
  - Reflexive: $a \sqsubseteq a$
  - Transitive: $a \sqsubseteq b$ and $b \sqsubseteq c$ implies $a \sqsubseteq c$
  - Antisymmetric: $a \sqsubseteq b$ and $b \sqsubseteq a$ implies $a = b$
- Examples: the subset relation, the divisibility relation on the naturals, ...
- Place a partial order on $\mathbb{Z} \cup \{\bot, \top\}$: $\bot \sqsubseteq n \sqsubseteq \top$ (most information to least information)
- Lift the ordering to constant environments: $f \sqsubseteq g$ iff $f(x) \sqsubseteq g(x)$ for all x
  - $f \sqsubseteq g$: f is a "better" constant environment than g
  - f sends x to $\top$ implies g sends x to $\top$
- The merge operation $\sqcup$ is the *least upper bound* in this order:
  - $f_1 \sqsubseteq (f_1 \sqcup f_2)$ and $f_2 \sqsubseteq (f_1 \sqcup f_2)$
  - For any f' such that $f_1 \sqsubseteq f'$ and $f_2 \sqsubseteq f'$, we have $(f_1 \sqcup f_2) \sqsubseteq f'$

Constant propagation as a constraint system

- Let $G = (N, E, s)$ be a control flow graph.
- For each basic block *bb* $\in N$, associate two constant environments $\mathbf{IN}[bb]$ and $\mathbf{OUT}[bb]$
    - $\mathbf{IN}[bb]$ is the constant environment at the *entry* of *bb*
    - $\mathbf{OUT}[bb]$ is the constant environment at the *exit* of *bb*

Constant propagation as a constraint system

- Let $G = (N, E, s)$ be a control flow graph.
- For each basic block *bb* $\in N$, associate two constant environments $\mathbf{IN}[bb]$ and $\mathbf{OUT}[bb]$
  - $\mathbf{IN}[bb]$ is the constant environment at the *entry* of *bb*
  - $\mathbf{OUT}[bb]$ is the constant environment at the *exit* of *bb*
- Say that the assignment $\mathbf{IN}$, **OUT** is **conservative** if
  1. $\mathbf{IN}[s]$ assigns each variable $\top$
  2. For each node *bb* $\in N$,
  $$\mathbf{OUT}[bb] \sqsupseteq post(bb, \mathbf{IN}[bb])$$
  3. For each edge $src \to dst \in E$,
  $$\mathbf{IN}[dst] \sqsupseteq \mathbf{OUT}[src]$$

Constant propagation as a constraint system

- Let $G = (N, E, s)$ be a control flow graph.
- For each basic block *bb* $\in N$, associate two constant environments $\mathbf{IN}[bb]$ and $\mathbf{OUT}[bb]$
  - $\mathbf{IN}[bb]$ is the constant environment at the *entry* of *bb*
  - $\mathbf{OUT}[bb]$ is the constant environment at the *exit* of *bb*
- Say that the assignment $\mathbf{IN}$, **OUT** is **conservative** if
  1. $\mathbf{IN}[s]$ assigns each variable $\top$
  2. For each node *bb* $\in N$,
  $$\mathbf{OUT}[bb] \sqsupseteq post(bb, \mathbf{IN}[bb])$$
  3. For each edge *src* $\rightarrow$ *dst* $\in E$,
  $$\mathbf{IN}[dst] \sqsupseteq \mathbf{OUT}[src]$$
- Fact: if $\mathbf{IN}$, **OUT** is conservative, then
  - If $\mathbf{IN}[bb](x) = n$, then whenever program execution reaches *bb* entry, the value of x is n
  - If $\mathbf{IN}[bb](x) = \bot$, then program execution cannot reach *bb*
  - Similarly for $\mathbf{OUT}$

- Think of $\mathbf{IN}[bb]$ and $\mathbf{OUT}[bb]$ as *variables* in a constraint system.
- The constraints may have multiple solutions
  - Recall: when constant environment sends a variables x to a constant (not $\top$), can replace reads to x with that constant
  - More constant assigments $\Rightarrow$ more optimization

- Think of $\mathbf{IN}[bb]$ and $\mathbf{OUT}[bb]$ as *variables* in a constraint system.
- The constraints may have multiple solutions
  - Recall: when constant environment sends a variables x to a constant (not $\top$), can replace reads to x with that constant
  - More constant assigments $\Rightarrow$ more optimization
- Want *least* conservative assignment
  1. $\mathbf{IN}, \mathbf{OUT}$ is conservative
  2. If $\mathbf{IN}', \mathbf{OUT}'$ is a conservative assignment, then for any *bb* we have
     - $\mathbf{IN}[bb] \sqsubseteq \mathbf{IN}'[bb]$
     - $\mathbf{OUT}[bb] \sqsubseteq \mathbf{OUT}'[bb]$

Computing the least conservative assignment of constant environments

- Initialize $\mathbf{IN}[s]$ to the constant environment that sends every variable to $\top$ and $\mathbf{OUT}[s]$ to the constant environment that sends every variable to $\bot$.
- Initialize $\mathbf{IN}[bb]$ and $\mathbf{OUT}[bb]$ to the constant environment that sends every variable to $\bot$ for every other basic block

Computing the least conservative assignment of constant environments

- Initialize $\mathbf{IN}[s]$ to the constant environment that sends every variable to $\top$ and $\mathbf{OUT}[s]$ to the constant environment that sends every variable to $\bot$.
- Initialize $\mathbf{IN}[bb]$ and $\mathbf{OUT}[bb]$ to the constant environment that sends every variable to $\bot$ for every other basic block
- Choose a constraint that is *not* satisfied by $\mathbf{IN}, \mathbf{OUT}$
  - If there is basic block *bb* with $\mathbf{OUT}[bb] \not\sqsupseteq post(bb, \mathbf{IN}[bb])$, then set
$$\mathbf{OUT}[bb] := post(bb, \mathbf{IN}[bb])$$
  - If there is an edge $src \rightarrow dst \in E$ with $\mathbf{IN}[dst] \not\sqsupseteq \mathbf{OUT}[src]$, then set
$$\mathbf{IN}[dst] := \mathbf{IN}[dst] \sqcup \mathbf{OUT}[src]$$
- Terminate when all constraints are satisfied.

Computing the least conservative assignment of constant environments

- Initialize $\mathbf{IN}[s]$ to the constant environment that sends every variable to $\top$ and $\mathbf{OUT}[s]$ to the constant environment that sends every variable to $\bot$.
- Initialize $\mathbf{IN}[\mathit{bb}]$ and $\mathbf{OUT}[\mathit{bb}]$ to the constant environment that sends every variable to $\bot$ for every other basic block
- Choose a constraint that is *not* satisfied by $\mathbf{IN}, \mathbf{OUT}$
  - If there is basic block *bb* with $\mathbf{OUT}[\mathit{bb}] \not\sqsupseteq \mathit{post}(\mathit{bb}, \mathbf{IN}[\mathit{bb}])$, then set

$$\mathbf{OUT}[\mathit{bb}] := \mathit{post}(\mathit{bb}, \mathbf{IN}[\mathit{bb}])$$

  - If there is an edge $\mathit{src} \to \mathit{dst} \in E$ with $\mathbf{IN}[\mathit{dst}] \not\sqsupseteq \mathbf{OUT}[\mathit{src}]$, then set

$$\mathbf{IN}[\mathit{dst}] := \mathbf{IN}[\mathit{dst}] \sqcup \mathbf{OUT}[\mathit{src}]$$

- Terminate when all constraints are satisfied.
- *This algorithm always converges on the least conservative assignment of constant environments*

Next week: *dataflow analysis*

- Framework for conservative analysis of program behavior
- *Worklist algorithm*: general algorithm for solving dataflow analysis problems