COS320: Compiling Techniques

Zak Kincaid

April 2, 2020
Optimization
Compiler phases (simplified)

Source text

→ Lexing

Token stream

→ Parsing

Abstract syntax tree

→ Translation

Intermediate representation

→ Optimization

→ Code generation

Assembly
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

\[ e \times 1 \rightarrow e \]

\[ 0 + e \rightarrow e \]

\[ 2 \times 3 \rightarrow 6 \]

\[ -(\neg e) \rightarrow e \]

\[ e \times 4 \rightarrow e \ll 2 \]

...
Loop unrolling

Idea: avoid branching by trading space for time.

```c
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;
}
```

→

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

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

```c
long trace (long *m, long n) {
    long i;
    long result = 0;
    long *next = m;
    for (i = 0; i < n; i++) {
        result += *next;
        next += n + i + 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 $\implies$ never perform an unsafe optimization, but may miss some safe optimizations.
```c
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 \( n_i \rightarrow n_j \in E \) 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:

\[
\begin{align*}
<\text{instr}> & ::= <\text{var}> = \text{add}<\text{opn}>, <\text{opn}> \\
& \quad | <\text{var}> = \text{mul}<\text{opn}>, <\text{opn}> \\
& \quad | <\text{var}> = \text{opn} \\
<\text{opn}> & ::= <\text{int}> | <\text{var}> \\
<\text{block}> & ::= <\text{instr}><\text{block}> | <\text{term}> \\
<\text{term}> & ::= \text{blez}<\text{opn}>, <\text{label}>, <\text{label}> \\
<\text{program}> & ::= <\text{program}> <\text{label}> : <\text{block}> | <\text{block}>
\end{align*}
\]

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 compute expressions at compile time to save on run time

\[
\begin{align*}
x &= \text{add } 1, 2 \\
y &= \text{mul } x, 11 \\
z &= \text{add } x, y
\end{align*}
\]
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 compute expressions at compile time to save on run time

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

- $x = \text{add} \ 1, \ 2$
- $y = \text{mul} \ x, \ 11$
- $z = \text{add} \ x, \ y$
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 compute expressions at compile time to save on run time

\[
\begin{align*}
\{x \mapsto T, y \mapsto T, z \mapsto T\} \\
\{x \mapsto 3, y \mapsto T, z \mapsto T\}
\end{align*}
\]

\[
\begin{align*}
x &= \text{add} \ 1, \ 2 \\
y &= \text{mul} \ x, \ 11 \\
z &= \text{add} \ x, \ y
\end{align*}
\]
Constant propagation

- The goal of constant propagation: determine at each instruction $I$ a constant environment $\langle I \rangle$
- 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 compute expressions at compile time to save on run time

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

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

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

$x = \text{add } 1, 2$

$y = \text{mul } x, 11$

$z = \text{add } x, y$
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 compute expressions at compile time to save on run time

\[
\begin{align*}
\{x &\mapsto T, y \mapsto T, z \mapsto T\} \\
\{x &\mapsto 3, y \mapsto T, z \mapsto T\} &\quad x = 3 \\
\{x &\mapsto 3, y \mapsto 33, z \mapsto T\} &\quad y = \text{mul } x, 11 \\
&\quad z = \text{add } x, y
\end{align*}
\]
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 compute expressions at compile time to save on run time

\[
\begin{align*}
\{x &\mapsto T, y \mapsto T, z \mapsto T \} \\
\{x &\mapsto 3, y \mapsto T, z \mapsto T \} \\
\{x &\mapsto 3, y \mapsto 33, z \mapsto T \}
\end{align*}
\]

\[
\begin{align*}
x &= 3 \\
y &= 33 \\
z &= \text{add } x, y
\end{align*}
\]
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 compute expressions at compile time to save on run time

\[
\begin{align*}
\{x \mapsto \top, y \mapsto \top, z \mapsto \top\} \\
\{x \mapsto 3, y \mapsto \top, z \mapsto \top\} \\
\{x \mapsto 3, y \mapsto 33, z \mapsto \top\}
\end{align*}
\]

\[
\begin{align*}
x &= 3 \\
y &= 33 \\
z &= 36
\end{align*}
\]
Propagating constants through instructions

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

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
  - \( x = \text{add}, \, \text{opn}_1, \, \text{opn}_2 \)
  - \( x = \text{mul}, \, \text{opn}_1, \, \text{opn}_2 \)
  - \( x = \text{opn} \)

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

- Define an evaluator for operands:

\[
\text{eval}(\text{opn}, \, C) = \begin{cases} 
C(\text{opn}) & \text{if opn is a variable} \\
\text{opn} & \text{if opn is an int}
\end{cases}
\]
Propagating constants through instructions

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

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

- Define an evaluator for operands:
  $$\text{eval}(\text{opn}, C) = \begin{cases} 
    C(\text{opn}) & \text{if opn is a variable} \\
    \text{opn} & \text{if opn is an int}
  \end{cases}$$

- Define an evaluator for instructions
  $$\text{post}(\text{instr}, C) = \begin{cases} 
    \bot & \text{if } C \text{ is } \bot \\
    C\{x \mapsto \text{eval}(\text{opn}, C)\} & \text{if instr is } x = \text{opn} \\
    C\{x \mapsto \top\} & \text{if } \text{eval}(\text{opn}_1, C) = \top \lor \text{eval}(\text{opn}_2, C) = \top \\
    C\{x \mapsto \text{eval}(\text{opn}_1, C) + \text{eval}(\text{opn}_2, C)\} & \text{if instr is } x = \text{add } \text{opn}_1, \text{opn}_2 \\
    C\{x \mapsto \text{eval}(\text{opn}_1, C) \times \text{eval}(\text{opn}_2, C)\} & \text{if instr is } x = \text{mul } \text{opn}_1, \text{opn}_2
  \end{cases}$$
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 $\text{instr}_1, \ldots, \text{instr}_n, \text{term}$.
  
  $\text{take post}(\text{block}, C) = \text{post}(\text{instr}_n, \ldots \text{post}(\text{instr}_1, C))$
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:

```
x = 0
y = x+1
z = y+2
br tgt
```

```
x = 0
y = 0
br tgt
```

\[
\begin{align*}
x &\mapsto 0, y \mapsto 1, z \mapsto 3 \\
\end{align*}
\]

\[
\begin{align*}
x &\mapsto 0, y \mapsto 0, z \mapsto T
\end{align*}
\]
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:

\[
\begin{align*}
e \sqcup \bot &= \bot \sqcup e \\
(e_1 \sqcup e_2)(x) &= \begin{cases} e_1(x) & \text{if } e_1(x) = e_2(x) \\
\top & \text{otherwise} \end{cases}
\end{align*}
\]

\[
x = 0
y = x + 1
z = y + 2
\]
\[
\text{br tgt}
\]

\[
\{x \mapsto 0, y \mapsto 1, z \mapsto 3\}
\]

\[
x = 0
y = 0
\]
\[
\text{br tgt}
\]

\[
\{x \mapsto 0, y \mapsto 0, z \mapsto \top\}
\]

\[
\{x \mapsto 0, y \mapsto \top, z \mapsto \top\}
\]
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:
    - $e \sqcup \bot = \bot \sqcup e = e$
    - $(e_1 \sqcup e_2)(x) = \begin{cases} e_1(x) & \text{if } e_1(x) = e_2(x) \\ \top & \text{otherwise} \end{cases}$

\[
\begin{align*}
x &= 0 \\
y &= x + 1 \\
z &= y + 2 \\
\br\ tgt
\end{align*}
\]
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 integers, ...

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:

\( t_1 \sqsubseteq (t_1 \sqcup t_2) \) and \( t_2 \sqsubseteq (t_1 \sqcup t_2) \)

For any type \( t' \) such that \( t_1 \sqsubseteq t' \) and \( t_2 \sqsubseteq t' \), we have \( (t_1 \sqcup t_2) \sqsubseteq t' \)
• 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 integers, ...

• 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 integers, ...

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:
- $t_1 \sqsubseteq (t_1 \sqcup t_2)$ and $t_2 \sqsubseteq (t_1 \sqcup t_2)$
- For any type $t'$ such that $t_1 \sqsubseteq t'$ and $t_2 \sqsubseteq t'$, we have $(t_1 \sqcup t_2) \sqsubseteq t'$
• 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 integers, ...

• 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:
  • \( t_1 \sqsubseteq (t_1 \sqcup t_2) \) and \( t_2 \sqsubseteq (t_1 \sqcup t_2) \)
  • For any type \( t' \) such that \( t_1 \sqsubseteq t' \) and \( t_2 \sqsubseteq t' \), we have \( (t_1 \sqcup t_2) \sqsubseteq t' \)
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 $IN[bb]$ and $OUT[bb]$
  - $IN[bb]$ is the constant environment at the *entry* of $bb$
  - $OUT[bb]$ is the constant environment at the *exit* of $bb$

---

- Fact: if $IN, OUT$ is conservative,
  - If $IN[bb](x) = n$, then whenever program execution reaches $bb$ entry, the value of $x$ is $n$
  - If $IN[bb](x) = \perp$, then program execution cannot reach $bb$
  - Similarly for $OUT$
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 $IN[bb]$ and $OUT[bb]$
  - $IN[bb]$ is the constant environment at the entry of $bb$
  - $OUT[bb]$ is the constant environment at the exit of $bb$
- Say that the assignment $IN$, $OUT$ is conservative if
  1. $IN[s]$ assigns each variable $\top$
  2. For each node $bb \in V$, $OUT[bb] \sqsubseteq post(bb, IN[bb])$
  3. For each edge $src \rightarrow dst \in E$, $IN[dst] \sqsubseteq OUT[src]$
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 $\text{IN}[bb]$ and $\text{OUT}[bb]$
  - $\text{IN}[bb]$ is the constant environment at the entry of $bb$
  - $\text{OUT}[bb]$ is the constant environment at the exit of $bb$
- Say that the assignment $\text{IN}$, $\text{OUT}$ is conservative if
  1. $\text{IN}[s]$ assigns each variable $\top$
  2. For each node $bb \in V$,
     $$\text{OUT}[bb] \sqsubseteq \text{post}(bb, \text{IN}[bb])$$
  3. For each edge $src \rightarrow dst \in E$,
     $$\text{IN}[dst] \sqsubseteq \text{OUT}[src]$$
- Fact: if $\text{IN}$, $\text{OUT}$ is conservative, then
  - If $\text{IN}[bb](x) = n$, then whenever program execution reaches $bb$ entry, the value of $x$ is $n$
  - If $\text{IN}[bb](x) = \bot$, then program execution cannot reach $bb$
  - Similarly for $\text{OUT}$
• Payoff: when constant environment sends a variables $x$ to a constant (not $\top$), can replace reads to $x$ with that constant
• More constant assignments $\Rightarrow$ more optimization
• Payoff: when constant environment sends a variables $x$ to a constant (not $\top$), can replace reads to $x$ with that constant

• More constant assignments $\Rightarrow$ more optimization

• Want *least* conservative assignment
  
  1. $\text{IN}, \text{OUT}$ is conservative
  2. If $\text{IN}'$, $\text{OUT}'$ is a conservative assignment, then for any $bb$ we have

    - $\text{IN}[bb] \subseteq \text{IN}'[bb]$
    - $\text{OUT}[bb] \subseteq \text{OUT}'[bb]$
Computing the least conservative assignment of constant environments

- Initialize $\text{IN}[s]$ to the constant environment that sends every variable to $\top$ and $\text{OUT}[s]$ to the constant environment that sends every variable to $\bot$.
- Initialize $\text{IN}[bb]$ and $\text{OUT}[bb]$ to the constant environment that sends every variable to $\bot$ for every other basic block.
Computing the least conservative assignment of constant environments

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

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