Lecture 11: Data Flow Analysis III
Review: Compiler Middle End (Optimizer)

- Front end works from the syntax of the source code
- Rest of the compiler works from intermediate representation (IR)
  - Analyzes IR to learn about the code
  - Transforms IR to improve final code
  - Examples
    1. Parallelization
    2. Redundancy elimination
    3. Data flow analysis (DFA) framework
    4. Single static assignment (SSA)
An expression $x+y$ is redundant if and only if, along every path from the procedure’s entry, it has been evaluated, and its constituent subexpressions ($x$ & $y$) have not been re-defined.

If the compiler can prove that an expression is redundant

- It can preserve the results of earlier evaluations
- It can replace the current evaluation with a reference

Two pieces to the problem

- Proving that $x+y$ is redundant
- Rewriting the code to eliminate the redundant evaluation
Review

Redundancy Elimination

- Local Value Numbering (LVN)
- Superlocal Value Numbering (SVN)
- Dominator Value Numbering (DVN)
- Global Common Subexpression Elimination (GCSE)
Local Value Numbering (LVN)

Find redundant operation within a block

LVN finds 2 redundant ops
LVN misses 8 redundant ops!
Local Value Numbering (LVN)

Find redundant operation within a block

LVN finds 2 redundant ops
LVN misses 8 redundant ops!
Review: Local Value Numbering

Local Value Numbering (LVN)
Find redundant operation within a block

LVN finds 2 redundant ops
LVN misses 8 redundant ops!
Review: Superlocal Value Numbering

Local Value Numbering (LVN)
Find redundant operation within a block

Superlocal Value Numbering (SVN)
Find redundant operation within a EBB
Local Value Numbering (LVN)
Find redundant operation within a block

Superlocal Value Numbering (SVN)
Find redundant operation within a EBB

An Extended Basic Block (EBB)
Set of blocks $B_1, B_2, \ldots, B_n$
- $B_1$ is either the entry node to the procedure, or it has $> 1$ pred.
- All other $B_i$ have 1 pred. & that pred. is in the EBB

Three EBBs in this CFG
1. $\{A, B, C, D, E\}$
2. $\{F\}$
3. $\{G\}$
Review: Superlocal Value Numbering

Local Value Numbering (LVN)
Find redundant operation within a block

Superlocal Value Numbering (SVN)
Find redundant operation within a EBB

An Extended Basic Block (EBB)
Set of blocks $B_1, B_2, \ldots, B_n$
- $B_1$ is either the entry node to the procedure, or it has $> 1$ pred.
- All other $B_i$ have 1 pred. & that pred. is in the EBB

Three EBBs in this CFG
1. \{ A, B, C, D, E \}
2. \{ F \}
3. \{ G \}
Review: Superlocal Value Numbering

Local Value Numbering (LVN)
Find redundancy within a block

Superlocal Value Numbering (SVN)
Find redundancy within a EBB

SVN finds 5 redundant ops.
SVN misses 5 redundant ops!
Review: Superlocal Value Numbering

**Local Value Numbering (LVN)**
Find redundancy within a block

**Superlocal Value Numbering (SVN)**
Find redundancy within a EBB

**Dominator Value Numbering (DVN)**
Find redundancy in IDOM-tree
An expression $x+y$ is redundant if and only if, along every path from the procedure’s entry, it has been evaluated, and its constituent subexpressions $(x \& y)$ have not been re-defined.

We need to understand the properties of those paths

- Specifically, what blocks occur on any path from entry to $p$
- If we can find those blocks, we can use their hash tables

For example:

- A is on every path to any other node; and C is on every path to D, E, and F.
- Only A is on every path to G.
Review: Dominance in a Control Flow Graph

Definition

In a flow graph, x dominates y if and only if every path from the entry node of the control-flow graph to y includes x

• By definition, x dominates x itself
• We associate a DOM set with each node
• DOM(x) contains the set of nodes that dominate x
• |DOM(x)| ≥ 1

<table>
<thead>
<tr>
<th>Block</th>
<th>DOM</th>
<th>IDOM</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
<td>—</td>
</tr>
<tr>
<td>B</td>
<td>A,B</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>A,C</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>A,C,D</td>
<td>C</td>
</tr>
<tr>
<td>E</td>
<td>A,C,E</td>
<td>C</td>
</tr>
<tr>
<td>F</td>
<td>A,C,F</td>
<td>C</td>
</tr>
<tr>
<td>G</td>
<td>A,G</td>
<td>A</td>
</tr>
</tbody>
</table>
1. Find blocks $DOM(p)$ occur on any path from entry to $p$.

2. Use hash table of $DOM(p)$ to remove redundancy.

**Local Value Numbering (LVN)**
Find redundancy within a block

**Superlocal Value Numbering (SVN)**
Find redundancy within a EBB

**Dominator Value Numbering (DVN)**
Find redundancy in IDOM-tree

1. Find blocks $DOM(p)$ occur on any path from entry to $p$.
2. Use hash table of $DOM(p)$ to remove redundancy
Review: Superlocal Value Numbering

Local Value Numbering (LVN)
Find redundancy within a block

Superlocal Value Numbering (SVN)
Find redundancy within a EBB

Dominator Value Numbering (DVN)
Find redundancy in IDOM-tree

<table>
<thead>
<tr>
<th>Block</th>
<th>DOM</th>
<th>IDOM</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
<td>—</td>
</tr>
<tr>
<td>B</td>
<td>A,B</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>A,C</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>A,C,D</td>
<td>C</td>
</tr>
<tr>
<td>E</td>
<td>A,C,E</td>
<td>C</td>
</tr>
<tr>
<td>F</td>
<td>A,C,F</td>
<td>C</td>
</tr>
<tr>
<td>G</td>
<td>A,G</td>
<td>A</td>
</tr>
</tbody>
</table>
Review: Superlocal Value Numbering

Local Value Numbering (LVN)
Find redundancy within a block

Superlocal Value Numbering (SVN)
Find redundancy within a EBB

Dominator Value Numbering (DVN)
Find redundancy in IDOM-tree

<table>
<thead>
<tr>
<th>Block</th>
<th>DOM</th>
<th>IDOM</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
<td>—</td>
</tr>
<tr>
<td>B</td>
<td>A,B</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>A,C</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>A,C,D</td>
<td>C</td>
</tr>
<tr>
<td>E</td>
<td>A,C,E</td>
<td>C</td>
</tr>
<tr>
<td>F</td>
<td>A,C,F</td>
<td>C</td>
</tr>
<tr>
<td>G</td>
<td>A,G</td>
<td>A</td>
</tr>
</tbody>
</table>
Review: Superlocal Value Numbering

Local Value Numbering (LVN)
Find redundancy within a block

Superlocal Value Numbering (SVN)
Find redundancy within an EBB

Dominator Value Numbering (DVN)
Find redundancy in IDOM-tree

<table>
<thead>
<tr>
<th>Block</th>
<th>DOM</th>
<th>IDOM</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
<td>—</td>
</tr>
<tr>
<td>B</td>
<td>A,B</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>A,C</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>A,C,D</td>
<td>C</td>
</tr>
<tr>
<td>E</td>
<td>A,C,E</td>
<td>C</td>
</tr>
<tr>
<td>F</td>
<td>A,C,F</td>
<td>C</td>
</tr>
<tr>
<td>G</td>
<td>A,G</td>
<td>A</td>
</tr>
</tbody>
</table>
Review: Superlocal Value Numbering

Local Value Numbering (LVN)
Find redundancy within a block

Superlocal Value Numbering (SVN)
Find redundancy within an EBB

Dominator Value Numbering (DVN)
Find redundancy in IDOM-tree

Block | DOM | IDOM
--- | --- | ---
A | A | —
B | A,B | A
C | A,C | A
D | A,C,D | C
E | A,C,E | C
F | A,C,F | C
G | A,G | A

Find blocks \( \text{DOM}(p) \) occur on any path from entry to \( p \).

Use hash table of \( \text{DOM}(p) \) to remove redundancy.

\[
m \leftarrow a + b \\
n \leftarrow a + b \\
p \leftarrow c + d \\
r \leftarrow c + d \\
e \leftarrow b + 18 \\
s \leftarrow a + b \\
u \leftarrow e + f \\
e \leftarrow a + 17 \\
t \leftarrow c + d \\
u \leftarrow e + f \\
v \leftarrow a + b \\
w \leftarrow c + d \\
x \leftarrow e + f \\
y \leftarrow a + b \\
z \leftarrow c + d \\
q \leftarrow a + b \\
r \leftarrow c + d \\
q \leftarrow a + b \\
r \leftarrow c + d \\
q \leftarrow a + b \\
r \leftarrow c + d 
\]
Value Numbering

Local Value Numbering (LVN)
Find redundancy within a block

Superlocal Value Numbering (SVN)
Find redundancy within a EBB

Dominator Value Numbering (DVN)
Find redundancy in IDOM-tree

DVN finds 8 redundant ops.
DVN misses 2 redundant ops!

Another shortcoming of DVN
No loop-carried redundancies
Value Numbering

Local Value Numbering (LVN)
Find redundancy within a block

Superlocal Value Numbering (SVN)
Find redundancy within a EBB

Dominator Value Numbering (DVN)
Find redundancy in IDOM-tree

Global Common Subexpression Elimination (GCSE)
Find redundancy in all basic blocks
Global Redundancy Elimination

**Availability definition:**

An expression is *available* on entry to block $b$ if and only if, along every path from $n_0$ to $b$, it has been evaluated and none of its constituent subexpressions has been re-defined.
Annotate each block $b$ with a set $\text{AVAIL}(b)$

$x+y \in \text{AVAIL}(b)$ iff $x+y$ is available at the entry to block $b$

$$\text{AVAIL}(b) = \bigcap_{x \in \text{predecessors}(b)} (\text{DEEXPR}(x) \cup (\text{AVAIL}(x) - \text{EXPRKILL}(x)))$$

**DEEXPR($x$):** the set of expressions defined in block $x$ and not killed before the end of block $x$

*(downward exposed expressions)*

**EXPRKILL($x$):** the set of expressions killed in block $x$

*(killed expressions)*

**Initial Conditions:**

- $\text{AVAIL}(n_0) = \emptyset$
- $\text{AVAIL}(x) = \{ \text{all expressions} \}$

```
<table>
<thead>
<tr>
<th>a = b + c</th>
<th>DEExpr: {b+c}</th>
</tr>
</thead>
<tbody>
<tr>
<td>f = a</td>
<td></td>
</tr>
<tr>
<td>u = f + e</td>
<td></td>
</tr>
<tr>
<td>l = b + u</td>
<td></td>
</tr>
<tr>
<td>EXPRKILL: {f+e}</td>
<td></td>
</tr>
<tr>
<td>AVAIL: {b+c, b+u}</td>
<td></td>
</tr>
</tbody>
</table>
```

```
Computing Availability

To compute AVAIL sets:

1. Build a control-flow graph (CFG)
2. For each block $b$ in the CFG, compute $DEEXPR$ and $EXPRKILL$
3. Solve the equations for AVAIL
   I. Use a data-flow solver (e.g., round-robin iterative solver)
   II. The AVAIL equations are well-behaved (unique fixed point, fast convergence)

\[
\text{AVAIL}(b) = \bigcap_{x \in \text{predecessors}(b)} (\text{DEEXPR}(x) \cup (\text{AVAIL}(x) - \text{EXPRKILL}(x)))
\]
Example

\[
\begin{align*}
\text{AVAIL}(A) &= \emptyset \\
\text{AVAIL}(B) &= \{a+b\} \cup (\emptyset \cap \text{all}) \\
&= \{a+b\} \\
\text{AVAIL}(C) &= \{a+b\} \\
\text{AVAIL}(D) &= \{a+b, c+d\} \cup (\{a+b, c+d\} \cap \text{all}) \\
&= \{a+b, c+d\} \\
\text{AVAIL}(E) &= \{a+b, c+d\} \\
\text{AVAIL}(F) &= \left[\{b+18, a+b, e+f\} \cup \left(\{a+b, c+d\} \cap \{\text{all} - e+f\}\right)\right] \\
&\quad \cap \left[\{a+17, c+d, e+f\} \cup \left(\{a+b, c+d\} \cap \{\text{all} - e+f\}\right)\right] \\
&= \{a+b, c+d, e+f\} \\
\text{AVAIL}(G) &= \left[\{c+d\} \cup (\{a+b\} \cap \text{all})\right] \\
&\quad \cap \left[\{a+b, c+d, e+f\} \cup (\{a+b, c+d, e+f\} \cap \text{all})\right] \\
&= \{a+b, c+d\}
\end{align*}
\]

\[
\begin{array}{c|c|c|c|c|c|c|c}
\text{DEExpr} & A & B & C & D & E & F & G \\
\hline
\text{a+b} & c+d & a+b, c+d & b+18, a+b, e+f & a+17, c+d, e+f & a+b, c+d, e+f & a+b, c+d \\
\text{EXPRKill} & \{} & \{} & \{} & e+f & e+f & \{} & \{}
\end{array}
\]

\[
\begin{align*}
A & \quad m \leftarrow a + b \\
& \quad n \leftarrow a + b \\
B & \quad p \leftarrow c + d \\
& \quad r \leftarrow c + d \\
C & \quad q \leftarrow a + b \\
& \quad r \leftarrow c + d \\
D & \quad e \leftarrow b + 18 \\
& \quad s \leftarrow a + b \\
& \quad u \leftarrow e + f \\
E & \quad e \leftarrow a + 17 \\
& \quad t \leftarrow c + d \\
& \quad u \leftarrow e + f \\
F & \quad v \leftarrow a + b \\
& \quad w \leftarrow c + d \\
& \quad x \leftarrow e + f \\
G & \quad y \leftarrow a + b \\
& \quad z \leftarrow c + d
\end{align*}
\]
Value Numbering

Local Value Numbering (LVN)
Find redundancy within a block

Superlocal Value Numbering (SVN)
Find redundancy within a EBB

Dominator Value Numbering (DVN)
Find redundancy in IDOM-tree

Global Common Subexpression Elimination (GCSE)
Find redundancy in all basic blocks

AVAIL(G) = \{a+b, c+d\}

AVAIL(F) = \{a+b, c+d, e+f\}
Value Numbering

Local Value Numbering (LVN)
Find redundancy within a block

Superlocal Value Numbering (SVN)
Find redundancy within a EBB

Dominator Value Numbering (DVN)
Find redundancy in IDOM-tree

Global Common Subexpression Elimination (GCSE)
Find redundancy in all basic blocks

GCSE finds 10 redundant ops.
GCSE misses 0 redundant ops!
Summary

Redundancy Elimination

- Local Value Numbering (LVN)
  - Finds redundancy, constants, & algebraic identities in a block
- Superlocal Value Numbering (SVN)
  - Extends local value numbering to EBBs
  - Use SSA-like name space to simplify bookkeeping
- Dominator Value Numbering (DVN)
  - Extends scope to “almost” global (no back edges)
  - Use dominance information to handle join points in CFG
- Global Common Subexpression Elimination (GCSE)
  - Calculate available expression at every basic block
  - Use data-flow analysis to calculate available expressions
Review: Compiler Middle End (Optimizer)

- Front end works from the syntax of the source code
- Rest of the compiler works from intermediate representation (IR)
  - Analyzes IR to learn about the code
  - Transforms IR to improve final code
  - Examples
    1. Parallelization
    2. Redundancy elimination
    3. Data flow analysis (DFA) framework
    4. Single static assignment (SSA)
Data Flow Analysis

• Almost always involves building a graph
  - Global problems ⇒ control-flow graphs (or derivative)
  - Whole program problems ⇒ call graph (or derivative)

• Usually formulated as simultaneous equations over sets of values
  - Sets attached to nodes and/or edges
  - Semi-lattice to describe values
  - Can be solved with an iterative fixed-point algorithm

• Desired result is usually meet over all paths solution
  - “What is true on every path from the entry?”
  - “Can this happen on any path from the entry?”
  - Related to the safety of optimization

A collection of techniques for compile-time reasoning about the run-time flow of values
DFA Examples We Have Seen

• Computing Dominance
• Available Expressions
Definition

*In a flow graph, $x$ dominates $y$ if and only if every path from the entry node of the control-flow graph to $y$ includes $x$*

- By definition, $x$ *dominates* $x$ itself
- We associate a DOM set with each node
- $\text{DOM}(x)$ contains the set of nodes that dominate $x$
- $|\text{DOM}(x)| \geq 1$

<table>
<thead>
<tr>
<th>Block</th>
<th>DOM</th>
<th>IDOM</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
<td>—</td>
</tr>
<tr>
<td>B</td>
<td>A,B</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>A,C</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>A,C,D</td>
<td>C</td>
</tr>
<tr>
<td>E</td>
<td>A,C,E</td>
<td>C</td>
</tr>
<tr>
<td>F</td>
<td>A,C,F</td>
<td>C</td>
</tr>
<tr>
<td>G</td>
<td>A,G</td>
<td>A</td>
</tr>
</tbody>
</table>
Review: Compute Dominance

Initially,
\[
\text{DOM}(n) = N, \quad \forall \ n \neq n_0 .
\]
\[
\text{DOM}(n_0) = \{ n_0 \}
\]

Data Flow Equation:
\[
\text{DOM}(n) = \{ n \} \cup (\cap_{p \in \text{preds}(n)} \text{DOM}(p))
\]

<table>
<thead>
<tr>
<th></th>
<th>B0</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
<th>B4</th>
<th>B5</th>
<th>B6</th>
<th>B7</th>
<th>B8</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>{0}</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>1</td>
<td>{0}</td>
<td>{0, 1}</td>
<td>{0, 1, 2}</td>
<td>{0, 1, 2, 3}</td>
<td>{0, 1, 2, 3, 4}</td>
<td>{0, 1, 5}</td>
<td>{0, 1, 5, 6}</td>
<td>{0, 1, 5, 7}</td>
<td>{0, 1, 5, 8}</td>
</tr>
<tr>
<td>2</td>
<td>{0}</td>
<td>{0, 1}</td>
<td>{0, 1, 2}</td>
<td>{0, 1, 3}</td>
<td>{0, 1, 3, 4}</td>
<td>{0, 1, 5}</td>
<td>{0, 1, 5, 6}</td>
<td>{0, 1, 5, 7}</td>
<td>{0, 1, 5, 8}</td>
</tr>
<tr>
<td>3</td>
<td>{0}</td>
<td>{0, 1}</td>
<td>{0, 1, 2}</td>
<td>{0, 1, 3}</td>
<td>{0, 1, 3, 4}</td>
<td>{0, 1, 5}</td>
<td>{0, 1, 5, 6}</td>
<td>{0, 1, 5, 7}</td>
<td>{0, 1, 5, 8}</td>
</tr>
</tbody>
</table>
Global Redundancy Elimination

**Availability definition:**

An expression is available on entry to block b if and only if, along every path from $n_0$ to b, it has been evaluated and none of its constituent subexpressions has been re-defined.
Annotate each block $b$ with a set $\text{AVAIL}(b)$

$x+y \in \text{AVAIL}(b)$ iff $x+y$ is available at the entry to block $b$

$\text{DEEXPR}(x)$: the set of expressions defined in block $x$ and not killed before the end of block $x$

$(\text{downward exposed expressions})$

$\text{EXPRKILL}(x)$: the set of expressions killed in block $x$

$(\text{killed expressions})$

Initial Conditions:

$\text{AVAIL}(n_0) = \emptyset$

$\text{AVAIL}(x) = \{ \text{all expressions} \}$

Data Flow Equation:

$\text{AVAIL}(b) = \bigcap_{x \in \text{predecessors}(b)} (\text{DEEXPR}(x) \cup (\text{AVAIL}(x) - \text{EXPRKILL}(x)))$
Data Flow Analysis Framework

- **Important Components** $D, V, \wedge, F$
  - Directions $D$: forward or backward
  - A semi-lattice including a domain $V$, and a meet operator $\wedge$
  - A transfer function family $F$ from $V$ to $V$ (effect of passing through a basic block), which should include functions good for boundary conditions.
Semilattices

- **V and **∧** form a semilattice if for all x, y, and z in V**
  - \( x \land x = x \) (idempotence)
  - \( x \land y = y \land x \) (commutativity)
  - \( x \land (y \land z) = (x \land y) \land z \) (associativity)
  - Top element \( \top \) such that for all \( x \), \( \top \land x = x \).
  - Bottom element (optional) \( \bot \) such that for all \( x \), \( \bot \land x = \bot \).
Example

- $V =$ power set of 4 nodes
  - With 4 nodes, there are $2^4$ possible sets
  - Think of each element $m$ in $V$ as a bit vector of the 4 nodes
- $\wedge =$ bitwise AND
- bitwise AND is idempotent, commutative, and associative
- What are the top and bottom elements?
  - Top element $\top$ such that for all $x$, $\top \wedge x = x$.
  - Bottom element (optional) $\bot$ such that for all $x$, $\bot \wedge x = \bot$
Example

- $V =$ power set of 4 nodes
  - With 4 nodes, there are $2^4$ possible sets
  - Think of each element $m$ in $V$ as a bit vector of the 4 nodes
- $\land =$ bitwise AND
- bitwise AND is idempotent, commutative, and associative
- What are the top and bottom elements?
  - Top element $\top$ such that for all $x$, $\top \land x = x$.
  - Bottom element (optional) $\bot$ such that for all $x$, $\bot \land x = \bot$

\[
\begin{array}{c}
\top = 1111 \\
\bot = 0000
\end{array}
\]
Partial Order for a Semilattice

- $x \leq y$ iff $x \land y = x$
- Also, $x < y$ iff $x \leq y$ and $x \neq y$
- $\leq$ is a partial order
  - I. $x \leq y$ and $y \leq z$ imply $x \leq z$
  - II. $x \leq y$ and $y \leq x$ iff $x = y$.

Proof: $x \land y = x$, $y \land x = y$
  - Since $x \land y = y \land x$, $x = y$. 

Example

- $V = \text{power set of 4 nodes}$
  - With 4 nodes, there are $2^4$ possible sets
  - Each element $m$ in $V$ as a bit vector of the 4 nodes
- $\wedge = \text{bitwise AND}$
Example

- $V = \text{power set of 4 nodes}$
  - With 4 nodes, there are $2^4$ possible sets
  - Each element $m$ in $V$ as a bit vector of the 4 nodes
- $\wedge = \text{bitwise AND}$
Example

- $V = \text{power set of 4 nodes}$
  - With 4 nodes, there are $2^4$ possible sets
  - Each element $m$ in $V$ as a bit vector of the 4 nodes
- $\wedge = \text{bitwise AND}$
Example

• $V =$ power set of 4 nodes
  - With 4 nodes, there are $2^4$ possible sets
  - Each element $m$ in $V$ as a bit vector of the 4 nodes
• $\land =$ bitwise AND
Transfer Functions

Effect of passing through a basic block

• F includes the identify function. $I(x) = x$ for all $x$ in $V$.
• F is closed under composition
  - $f, g$ in $F$; $h() = g(f())$; then $h$ also in $F$.
  - The concatenation of two blocks is one block
    Implication: transfer function for a block can be constructed from individual statements.
Example: Expression Availability

Annotate each block $b$ with a set $\text{AVAIL}(b)$

$x + y \in \text{AVAIL}(b)$ iff $x + y$ is available at the entry to block $b$

$\text{DEEXPR}(x)$: the set of expressions defined in block $x$ and not killed before the end of block $x$

($\text{downward exposed expressions}$)

$\text{EXPRKILL}(x)$: the set of expressions killed in block $x$

($\text{killed expressions}$)

Initial Conditions:

$\text{AVAIL}(n_0) = \emptyset$

$\text{AVAIL}(x) = \{ \text{all expressions} \}$

Data Flow Equation:

$\text{AVAIL}(b) = \bigcap_{x \in \text{predecessors}(b)} (\text{DEEXPR}(x) \cup (\text{AVAIL}(x) - \text{EXPRKILL}(x)))$
Semilattice Example for Expression Availability

Expression Availability

\((D, V, F, \land)\)

- \(D\): forward
- \(V\): all expressions
- \(F(m)\): \(\text{DEEXPR}(m) \cup (\text{AVAIL}(m) - \text{EXPRKILL}(m))\)
- \(\land = \text{set intersection} \cap\)

\[\text{EXPRKILL}: \{f+e\} \quad \text{DEExpr}: \{b+c\} \quad \text{AVAIL}: \{b+c, b+u\}\]

\[\text{AVAIL}(b) = \bigcap_{x \in \text{predecessors}(b)} (\text{DEEXPR}(x) \cup (\text{AVAIL}(x) - \text{EXPRKILL}(x)))\]
Review: Compute Dominance

Initially,

\[ \text{DOM}(n) = N, \ \forall \ n \neq n_0. \]
\[ \text{DOM}(n_0) = \{ n_0 \} \]

Data Flow Equation:

\[ \text{DOM}(n) = \{ n \} \cup (\cap_{p \ pred(n)} \text{DOM}(p)) \]

<table>
<thead>
<tr>
<th></th>
<th>DOM(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B0</td>
<td>{0}</td>
</tr>
<tr>
<td>B1</td>
<td>N</td>
</tr>
<tr>
<td>B2</td>
<td>N</td>
</tr>
<tr>
<td>B3</td>
<td>N</td>
</tr>
<tr>
<td>B4</td>
<td>N</td>
</tr>
<tr>
<td>B5</td>
<td>N</td>
</tr>
<tr>
<td>B6</td>
<td>N</td>
</tr>
<tr>
<td>B7</td>
<td>N</td>
</tr>
<tr>
<td>B8</td>
<td>N</td>
</tr>
</tbody>
</table>

- \(1\) \{0\} \{0,1\} \{0,1,2\} \{0,1,2,3\} \{0,1,2,3,4\} \{0,1,5\} \{0,1,5,6\} \{0,1,5,7\} \{0,1,5,8\}
- \(2\) \{0\} \{0,1\} \{0,1,2\} \{0,1,3\} \{0,1,3,4\} \{0,1,5\} \{0,1,5,6\} \{0,1,5,7\} \{0,1,5,8\}
- \(3\) \{0\} \{0,1\} \{0,1,2\} \{0,1,3\} \{0,1,3,4\} \{0,1,5\} \{0,1,5,6\} \{0,1,5,7\} \{0,1,5,8\}
Semilattice Example for Computing Dominance

Expression Availability

\((D, V, F, \land)\)

- \(D\): forward
- \(V\): all basic blocks (nodes in the CFG)
- \(F(m): \cup m\)
- \(\land\) = set intersection \(\cap\)

Initially,
\[
\text{DOM}(n) = N, \quad \forall \ n \neq n_0.
\]
\[
\text{DOM}(n_0) = \{ n_0 \}
\]

Data Flow Equation:
\[
\text{DOM}(n) = \{ n \} \cup (\cap_{p \ \text{preds}(n)} \text{DOM}(p))
\]

\(\top = \) the set of all nodes in the CFG

\(\bot = \) empty set
Semilattice Example for Computing Dominance

Termination Condition

- Each DOM set can contain at most $|N|$ elements — a finite number of elements
- New DOM sets are produced as the intersection of old DOM sets
  
  \[ |(D_1 \cap D_2)| \leq |D_1| \]
  
  \[ |(D_1 \cap D_2)| \leq |D_2| \]

- Intersection imposes an order on the $D_i$'s
  
  $D_1 \cap D_2$ is no larger than max($|D_1|$, $|D_2|$)

$D_i \geq D_j$ iff $D_i \cap D_j = D_j$

$D_i > D_j$ iff $D_i \geq D_j$ and $D_i \neq D_j$

Since the initial sets are finite, they can only shrink a finite number of times
  
  They must stop changing after a finite number of updates

The algorithm halts after a finite number of updates

Data Flow Equation:

$$\text{DOM}(n) = \{ n \} \cup (\bigcap_{p \in \text{preds}(n)} \text{DOM}(p))$$
Semilattice Example for Computing Dominance

Termination Condition

• Each DOM set can contain at most $|N|$ elements — a finite number of elements.
• New DOM sets are produced as the intersection of old DOM sets
  $| (D_1 \cap D_2) | \leq | D_1 |$
  $| (D_1 \cap D_2) | \leq | D_2 |$
• Intersection imposes an order on the $D_i$’s
  $D_1 \cap D_2$ is no larger than max($|D_1|$, $|D_2|$)
  $D_i \geq D_j$ iff $D_i \cap D_j = D_j$
  $D_i > D_j$ iff $D_i \geq D_j$ and $D_i \neq D_j$

Data Flow Equation:

$$\text{DOM}(n) = \{ n \} \cup (\bigcap_{p \in \text{preds}(n)} \text{DOM}(p))$$

Introduces the notion of a descending chain

Sequence of values $x_0, x_1, x_2, \ldots x_k$ where each $x_i$ is a set in $D$ and $x_i > x_{i+1}$, $1 \leq i < n$

If all descending chains are finite, then the algorithm halts

Since $D$ is finite, the chains must be finite
Review: Compiler Middle End (Optimizer)

- Front end works from the syntax of the source code
- Rest of the compiler works from intermediate representation (IR)
  - Analyzes IR to learn about the code
  - Transforms IR to improve final code
  - Examples
    1. Parallelization
    2. Redundancy elimination
    3. Data flow analysis (DFA) framework
    4. Single static assignment (SSA)
Review: Static Single Assignment (SSA) Form

Two principles
• Each name is defined by exactly one operation
• Each operand refers to exactly one definition

To reconcile these principles with real code
• Insert $\phi$-functions at merge points to reconcile name space
• Add subscripts to variable names for uniqueness
SSA Algorithm

1. Construct CFG
2. Compute Dom
3. Compute DF
4. Insert $\Phi$ functions
5. Rename
6. Resolve reaching definitions
7. “Deconstruct” SSA
SSA Construction — $\Phi$ insertion

Naive Insertion
• Insert $\Phi$-functions at every join for every name appearing in the CFG
• Solve reaching definitions
• Rename each use to the definition that reaches

Potential Issues
• Too many $\Phi$-functions inserted: not space or time efficient
• How to eliminate useless $\Phi$-functions?
  Only insert $\Phi$-function at the earliest meet points of two different values!
Static Single Assignment (SSA)

Meet operation
• \( x \leftarrow \Phi(y, z) \)
• \( \Phi \) placement

Naive SSA phi-insertion
• Put \( \Phi \) at every meet point
• How many \( \Phi \) functions are needed?
• How to eliminate useless \( \Phi \)-functions?

Only insert \( \Phi \)-function at the earliest meet points of two different values!
Control flow and data flow

How to identify the earliest meeting point of two values? Use dominance frontiers $DF(n)$

- A block $f$ is in $DF(n)$ if
  1. $n$ dominates a predecessor of $f$
  2. $n$ does not strictly dominate $f$

- The essential idea
  - $f$ is a join point
  - one of the predecessors of $f$ is dominated by $n$
SSA Algorithm

1. Construct CFG
2. Compute Dom
3. Compute DF
4. Insert $\Phi$ functions
5. Rename
6. Resolve reaching definitions
7. “Deconstruct” SSA
Review: Computing Dominance

Initially, $\text{DOM}(n) = N$, $\forall \ n \neq n_0$ and $\text{DOM}(n_0) = \{ n_0 \}$

$$\text{DOM}(n) = \{ n \} \cup (\cap_{p \in \text{preds}(n)} \text{DOM}(p))$$

Example:

$\text{DOM}(B_6) = \{ B_6 \} \cup \{ \text{DOM}(B_4) \cap \text{DOM}(B_5) \}$
Dominance Tree

Control Flow Graph

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOM</td>
<td>0</td>
<td>0, 1</td>
<td>0, 1, 2</td>
<td>0, 1, 3</td>
<td>0, 1, 3, 4</td>
<td>0, 1, 3, 5</td>
<td>0, 1, 3, 6</td>
<td>0, 1, 7</td>
</tr>
<tr>
<td>IDOM</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>
SSA Algorithm

1. Construct CFG
2. Compute Dom
3. **Compute DF**
4. Insert $\Phi$ functions
5. Rename
6. Resolve reaching definitions
7. “Deconstruct” SSA
Computing Dominance Frontier

Dominance Frontiers

- DF(n) is fringe just beyond the region n dominates
- m ∈ DF(n) : iff n ∉ (Dom(m) - {m}) but n ∈ DOM(p) for some p ∈ preds(m).

i.e., n dominates p

i.e., n doesn’t strictly dominate m
Computing Dominance Frontiers

- Only join points are in DF(n) for some n
- Leads to a simple, intuitive algorithm for computing dominance frontiers

For each join point $x$ (i.e., $|\text{preds}(x)| > 1$)
For each CFG predecessor $p$ of $x$
For each CFG node $n$ from $p$ to $\text{IDOM}(x)$ not including $\text{IDOM}(x)$ in the dominator tree,
adding $x$ to $\text{DF}(n)$ for each $n$ in the walk except $\text{IDOM}(x)$.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOM</td>
<td>0</td>
<td>1</td>
<td>0, 1, 2</td>
<td>0, 1, 3</td>
<td>0, 1, 3, 4</td>
<td>0, 1, 3, 5</td>
<td>0, 1, 3, 6</td>
<td>0, 1, 7</td>
</tr>
<tr>
<td>DF</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
SSA Algorithm

1. Construct CFG
2. Compute Dom
3. Compute DF
4. Insert $\Phi$ functions
5. Rename
6. Resolve reaching definitions
7. “Deconstruct” SSA
Example

\[ x \leftarrow \Phi(...) \]

\[ \text{Dominance Frontiers} \]

\[
\begin{array}{cccccccc}
\text{DOM} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
\text{DF} & - & 1 & 7 & 7 & 6 & 6 & 7 & 1 \\
\end{array}
\]

- DF(4) is \{6\}, so \( x \leftarrow \) in 4 forces \( \Phi \)-function in 6
- \( x \leftarrow \) in 6 forces \( \Phi \)-function in DF(6) = \{7\}
- \( x \leftarrow \) in 7 forces \( \Phi \)-function in DF(7) = \{1\}
- \( x \leftarrow \) in 1 forces \( \Phi \)-function in DF(1) = \{1\} (halt)
SSA Algorithm

1. Construct CFG
2. Compute Dom
3. Compute DF
4. Insert Φ functions
5. Rename
6. Resolve reaching definitions
7. “Deconstruct” SSA
Rename variables in a pre-order walk over dominator tree

Use an array of stacks, one stack per global name

Start with the root block b
• Generate unique names for each \( \Phi \) function and push them on the appropriate stacks
• Rewrite each operation in the block
  i. Rewrite uses of global names with current version from the stack
  ii. Rewrite definition by inventing & pushing new name
• Fill in \( \Phi \) function parameters of each successor blocks
• Recurse on b’s children in the dominator tree
• <on exit from b> pop names generated in b from stacks

1 counter per name for subscripts

Reset the state
Renaming Algorithm

Adding all the details ...

for each global name i
  counter[i] ← 0
  stack[i] ← \emptyset

call Rename(n_0)

NewName(n)
  i ← counter[n]
  counter[n] ← counter[n] + 1
  push n_i onto stack[n]
  return n_i

Rename(b)
  for each \( \Phi \)-function in b, \( x \leftarrow \Phi(\ldots) \)
    rename \( x \) as NewName(x)
  for each operation “\( x \leftarrow y \text{ op } z \)” in b
    rewrite \( y \) as top(stack[y])
    rewrite \( z \) as top(stack[z])
    rewrite \( x \) as NewName(x)
  for each successor of b in the CFG
    rewrite appropriate \( \Phi \) parameters
  for each successor s of b in dom. tree
    Rename(s)
  for each operation “\( x \leftarrow y \text{ op } z \)” in b
    pop(stack[x])
Assume a, b, c, & d defined before B₀

Example

Before processing B₀

Assume a, b, c, & d defined before B₀

Assume a, b, c, & d defined before B₀

Counters
Stacks

1 1 1 1 0

a b c d i

a₀ b₀ c₀ d₀

i has not been defined
Assume $a$, $b$, $c$, & $d$ defined before $B_0$

Example

End of $B_0$

Counters

Stacks

$\begin{array}{cccccc}
1 & 1 & 1 & 1 & 1 \\
a_0 & b_0 & c_0 & d_0 & i_0 
\end{array}$
Assume a, b, c, & d defined before B₀

Example

End of B₁

Counters

Stacks

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>B₀</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>B₁</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B₂</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B₃</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B₄</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B₅</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B₆</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

a₀ b₀ c₀ d₀ i₀

a₁ b₁ c₁ d₁ i₁

a₂ b₂ c₂
Assume a, b, c, & d defined before $B_0$

Example

End of $B_2$

Counter Stacks

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B_0$</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>$B_1$</td>
<td>$a_0$</td>
<td>$b_0$</td>
<td>$c_0$</td>
<td>$d_0$</td>
<td>$i_0$</td>
</tr>
<tr>
<td>$B_2$</td>
<td>$a_1$</td>
<td>$b_1$</td>
<td>$c_1$</td>
<td>$d_1$</td>
<td>$i_1$</td>
</tr>
<tr>
<td>$B_3$</td>
<td>$a_2$</td>
<td>$b_2$</td>
<td>$c_2$</td>
<td>$d_2$</td>
<td>$c_3$</td>
</tr>
</tbody>
</table>
Assume a, b, c, & d defined before B₀

Example

Before starting B₃

Counters

Stacks

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a₀</th>
<th>b₀</th>
<th>c₀</th>
<th>d₀</th>
<th>i₀</th>
</tr>
</thead>
<tbody>
<tr>
<td>a₁</td>
<td>b₁</td>
<td>c₁</td>
<td>d₁</td>
<td>i₁</td>
</tr>
<tr>
<td>a₂</td>
<td>c₂</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

i > 100

i ≤ 100
Example

Assume a, b, c, & d defined before B₀

End of B₃
Example

Assume $a, b, c, \& d$ defined before $B_0$
Example

Assume $a$, $b$, $c$, & $d$ defined before $B_0$

**Counters**

<table>
<thead>
<tr>
<th></th>
<th>$a$</th>
<th>$b$</th>
<th>$c$</th>
<th>$d$</th>
<th>$i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B_0$</td>
<td>4</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

**Stacks**

<table>
<thead>
<tr>
<th></th>
<th>$a_0$</th>
<th>$b_0$</th>
<th>$c_0$</th>
<th>$d_0$</th>
<th>$i_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B_0$</td>
<td>$a_1$</td>
<td>$b_1$</td>
<td>$c_1$</td>
<td>$d_1$</td>
<td>$i_1$</td>
</tr>
</tbody>
</table>

$y = a + b$
$z = c + d$
$i = i + 1$

$i > 100$

End of $B_5$
Assume a, b, c, & d defined before B₀
Example

Assume a, b, c, & d defined before B₀

Before B₇

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>B₀</td>
<td></td>
<td>i &gt; 100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B₁</td>
<td>a₁ ← Φ(a₀, a)</td>
<td>b₁ ← Φ(b₀, b)</td>
<td>c₁ ← Φ(c₀, c)</td>
<td>d₁ ← Φ(d₀, d)</td>
<td>i₁ ← Φ(i₀, i)</td>
</tr>
<tr>
<td>B₂</td>
<td>b₂ ← ***</td>
<td>c₂ ← ***</td>
<td>d₂ ← ***</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B₃</td>
<td>a₃ ← ***</td>
<td>d₃ ← ***</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B₄</td>
<td>d₄ ← ***</td>
<td>c₄ ← ***</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B₅</td>
<td>c₅ ← Φ(c₂, c₄)</td>
<td>b₅ ← ***</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B₆</td>
<td>d₅ ← Φ(d₄, d₃)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B₇</td>
<td>a ← Φ(a₂, a₃)</td>
<td>b ← Φ(b₂, b₁)</td>
<td>c ← Φ(c₃, c₅)</td>
<td>d ← Φ(d₄, d₂)</td>
<td></td>
</tr>
<tr>
<td>Counters</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>Stacks</td>
<td>a₀</td>
<td>b₀</td>
<td>c₀</td>
<td>d₀</td>
<td>i₀</td>
</tr>
</tbody>
</table>
Assume a, b, c, & d defined before B_0
After renaming

- Semi-pruned SSA form
- We’re done ...

Assume a, b, c, & d defined before B₀

Semi-pruned ⇒ only names live in 2 or more blocks are “global names”.

Example
SSA Algorithm

1. Construct CFG
2. Compute Dom
3. Compute DF
4. Insert $\Phi$ functions
5. Rename
6. Resolve reaching definitions
7. “Deconstruct” SSA
SSA Deconstruction

At some point, we need executable code
  • No machines implement \( \Phi \) operations
  • Need to fix up the flow of values

Basic idea
  • Insert copies to \( \Phi \)-function predecessors
  • Adds lots of copies
    Most of them coalesce away
Deconstruction Problem: Critical Edges

An edge whose source has multiple successors and whose destination has multiple predecessors

\[
\begin{align*}
B_0 & \quad i_0 \leftarrow \cdots \\
B_1 & \quad i_1 \leftarrow \Phi(i_0, i_2) \\
& \quad \cdots \quad \cdots \\
& \quad i_2 \leftarrow i_1 + 1 \\
& \quad z_0 \leftarrow i_1 + \cdots
\end{align*}
\]
Deconstruction Problem: Critical Edges

An edge whose source has multiple successors and whose destination has multiple predecessors
Reading

• Textbook
  Engineering a Compiler
  - Chapter 5.3.4
  - Chapter 5.5
  - Chapter 8.6.1
  - Chapter 9