Lecture 11: Data Flow Analysis III
(The lectures are based on the slides copyrighted by Keith Cooper and Linda Torczon from Rice University.)

Zheng (Eddy) Zhang
Rutgers University
Fall 2017, 11/28/2017
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)
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: Local Value Numbering

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: Superlocal Value Numbering

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

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

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

Superlocal Value Numbering (SVN)
Find redundant operation within an 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

A
m ← a + b
n ← a + b

B
p ← c + d
r ← c + d

C
q ← a + b
r ← c + d

e ← b + 18
s ← a + b
u ← e + f

D

e ← a + 17
t ← c + d
u ← e + f

E
v ← a + b
w ← c + d
x ← e + f

F

G
y ← a + b
z ← c + d
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
- \( \text{DOM}(x) \) 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>

\[
\begin{align*}
m & \leftarrow a + b \\
n & \leftarrow a + b \\
q & \leftarrow a + b \\
r & \leftarrow c + d \\
p & \leftarrow c + d \\
e & \leftarrow b + 18 \\
s & \leftarrow a + b \\
u & \leftarrow e + f \\
e & \leftarrow a + 17 \\
t & \leftarrow c + d \\
u & \leftarrow e + f \\
v & \leftarrow a + b \\
w & \leftarrow c + d \\
x & \leftarrow e + f \\
y & \leftarrow a + b \\
z & \leftarrow c + d
\end{align*}
\]
1. Find blocks $\text{DOM}(p)$ occur on any path from entry to $p$.

2. Use hash table of $\text{DOM}(p)$ 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 $\text{DOM}(p)$ occur on any path from entry to $p$.
2. Use hash table of $\text{DOM}(p)$ 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>
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 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 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>
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.
Data Flow Analysis for Expression Availability

Each block $b$ is annotated with a set $\text{AVAIL}(b)$:

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

**Data Flow Equation:**

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

**Initial Conditions:**

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

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

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

\[
\begin{align*}
\text{DEEXPR: } \{b+c\} \\
\text{EXPRKILL: } \{f+e\} \\
\text{AVAIL: } \{b+c, b+u\}
\end{align*}
\]
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)

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

AVAIL(A) = Ø
AVAIL(B) = \{a+b\} ∪ (Ø ∩ all)
    = \{a+b\}
AVAIL(C) = \{a+b\}
AVAIL(D) = \{a+b, c+d\} ∪ (\{a+b, c+d\} ∩ all)
    = \{a+b, c+d\}
AVAIL(E) = \{a+b, c+d\}
AVAIL(F) = [[\{b+18, a+b, e+f\} ∪ \{a+b, c+d\} ∩ all - e+f\}]]
    ∩ [[\{a+17, c+d, e+f\} ∩ \{a+b, c+d\} ∩ all - e+f\}]
    = \{a+b, c+d, e+f\}
AVAIL(G) = [[\{c+d\} ∪ (\{a+b\} ∩ all)]]
    ∩ [[\{a+b, c+d, e+f\} ∪ \{a+b, c+d, e+f\} ∩ all)]]
    = \{a+b, c+d\}

<table>
<thead>
<tr>
<th>DEExpr</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>a+b+c+d</td>
<td>a+b</td>
<td>c+d</td>
<td>a+b</td>
<td>b+18</td>
<td>a+b, e+f</td>
<td>a+17</td>
<td>c+d, e+f</td>
</tr>
<tr>
<td>a+b+c+d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ExprKill</td>
<td>{}</td>
<td>{}</td>
<td>{}</td>
<td>e+f</td>
<td>e+f</td>
<td>{}</td>
<td>{}</td>
</tr>
</tbody>
</table>

- m ← a + b
- n ← a + b
- q ← a + b
- r ← c + d
- e ← b + 18
- s ← a + b
- u ← e + f
- e ← a + 17
- t ← c + d
- u ← e + f
- v ← a + b
- w ← c + d
- x ← e + f
- y ← a + b
- z ← c + d

AVAIL(A) = Ø
AVAIL(B) = \{a+b\} ∪ (Ø ∩ all)
    = \{a+b\}
AVAIL(C) = \{a+b\}
AVAIL(D) = \{a+b, c+d\} ∪ (\{a+b, c+d\} ∩ all)
    = \{a+b, c+d\}
AVAIL(E) = \{a+b, c+d\}
AVAIL(F) = [[\{b+18, a+b, e+f\} ∪ \{a+b, c+d\} ∩ all - e+f\}]]
    ∩ [[\{a+17, c+d, e+f\} ∩ \{a+b, c+d\} ∩ all - e+f\}]
    = \{a+b, c+d, e+f\}
AVAIL(G) = [[\{c+d\} ∪ (\{a+b\} ∩ all)]]
    ∩ [[\{a+b, c+d, e+f\} ∪ \{a+b, c+d, e+f\} ∩ all)]]
    = \{a+b, 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

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

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

AVAIL(G) = \{a+b, c+d\}
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
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
- \( \text{DOM}(x) \) 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>

\[
\begin{align*}
\text{A} & \quad m \leftarrow a + b \\
\text{B} & \quad n \leftarrow a + b \\
\text{C} & \quad q \leftarrow a + b \\
\text{D} & \quad r \leftarrow c + d \\
\text{E} & \quad e \leftarrow b + 18 \\
\text{F} & \quad s \leftarrow a + b \\
\text{G} & \quad u \leftarrow e + f \\
\end{align*}
\]
Review: Compute Dominance

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

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.
Review: 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 $x$ (downward exposed expressions)

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

Initial Conditions:

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

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

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

Semilattices
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

- $V = \text{power set of 4 nodes}$
  - With 4 nodes, there are $2^4$ possible sets
  - Think of each element $m$ in $V$ as a bit vector of the 4 nodes
- $\land = \text{bitwise AND}$
- 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 $x$, $\bot \land x = \bot$

$$
\begin{align*}
\top &= 1111 \\
\bot &= 0000
\end{align*}
$$
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
- $\land = \text{bitwise AND}$
Example

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

($\textit{downward exposed expressions}$)

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

($\textit{killed expressions}$)

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
- \(F(m)\): \(DEEXPR(m) \cup (AVAIL(m) - EXPRKILL(m))\)
- \(\land = set\ intersection\ \cap\)

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

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

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

<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>0</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>
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 \(\cap\)

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

Initially,

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

\[
\text{DOM}(n_0) = \{ n_0 \}
\]

Data Flow Equation:

\[
\text{DOM}(n) = \{ n \} \cup (\cap_{p \in \text{preds}(n)} \text{DOM}(p))
\]
Semilattice Example for Computing Dominance

Termination Condition

- Each \( \text{DOM} \) set can contain at most \( |N| \) elements — a finite number of elements
- New \( \text{DOM} \) sets are produced as the intersection of old \( \text{DOM} \) 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

\[
\text{Data Flow Equation:} \\
\text{DOM}(n) = \{ n \} \cup (\bigcap_{p \in \text{preds}(n)} \text{DOM}(p))
\]
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 $\max(|D_1|, |D_2|)$
  \[ D_i \geq D_j \text{ iff } D_i \cap D_j = D_j \]
  \[ D_i > D_j \text{ iff } D_i \geq D_j \text{ and } D_i \neq D_j \]

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

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 Φ functions
5. Rename
6. Resolve reaching definitions
7. “Deconstruct” SSA
SSA Construction — Φ insertion

Naive Insertion
• Insert Φ-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 Φ-functions inserted: not space or time efficient
• How to eliminate useless Φ-functions?
  Only insert Φ-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 $\text{DOM}(n_0) = \{ n_0 \}$

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

Example:

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

<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>
Dominance Tree

Control Flow Graph

Dominance Tree

<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

<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>DF</td>
<td>-</td>
<td>1</td>
<td>7</td>
<td>7</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>1</td>
</tr>
</tbody>
</table>
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., |preds(x)| > 1)
   For each CFG predecessor p of x
      For each CFG node n from p to IDOM(x) not including IDOM(x) in the dominator tree,
         adding x to DF(n) for each n in the walk except 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>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</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

![Diagram of Dominance Frontiers]

<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>DF</td>
<td>-</td>
<td>1</td>
<td>7</td>
<td>7</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>1</td>
</tr>
</tbody>
</table>

- DF(4) is {6}, so $\leftarrow$ in 4 forces $\Phi$-function in 6
  - $\leftarrow$ in 6 forces $\Phi$-function in DF(6) = {7}
  - $\leftarrow$ in 7 forces $\Phi$-function in DF(7) = {1}
  - $\leftarrow$ 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
  - Rewrite uses of global names with current version from the stack
  - 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 $i$
  $\text{counter}[i] \leftarrow 0$
  $\text{stack}[i] \leftarrow \Phi$
  call Rename($n_0$)

NewName($n$)
  $i \leftarrow \text{counter}[n]$ 
  $\text{counter}[n] \leftarrow \text{counter}[n] + 1$
  push $n_i$ onto $\text{stack}[n]$
  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($\text{stack}[y]$)
    rewrite $z$ as top($\text{stack}[z]$)
    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[$x$])
Example

Before processing $B_0$

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

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

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

Counters

Stacks

1 1 1 1 0

$a_0$ $b_0$ $c_0$ $d_0$

$i$ has not been defined
Assume a, b, c, & d defined before B₀
Assume $a$, $b$, $c$, & $d$ defined before $B_0$

Example

End of $B_1$

Counts

Stacks

Assume $a$, $b$, $c$, & $d$ defined before $B_0$
Assume a, b, c, & d defined before B₀
Example

Before starting B₃

Assume a, b, c, & d defined before B₀

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>
<tr>
<td>a₀</td>
<td>b₀</td>
<td>c₀</td>
<td>d₀</td>
<td>i₀</td>
</tr>
<tr>
<td>a₁</td>
<td>b₁</td>
<td>c₁</td>
<td>d₁</td>
<td>i₁</td>
</tr>
<tr>
<td>a₂</td>
<td>b₂</td>
<td>c₂</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Assume $a$, $b$, $c$, & $d$ defined before $B_0$. 

End of $B_3$
Example

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

Assume a, b, c, & d defined before B₀

End of B₅

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>4</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>a₀</td>
<td>b₀</td>
<td>c₀</td>
<td>d₀</td>
<td>i₀</td>
</tr>
<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>d₃</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a₃</td>
<td>c₄</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Assume a, b, c, & d defined before B₀

Example

End of B₆

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>a₀</td>
<td>b₀</td>
<td>c₀</td>
<td>d₀</td>
<td>i₀</td>
</tr>
<tr>
<td>a₁</td>
<td>b₁</td>
<td>c₁</td>
<td>d₁</td>
<td>i₁</td>
</tr>
<tr>
<td>a₂</td>
<td>b₃</td>
<td>c₂</td>
<td>d₃</td>
<td></td>
</tr>
<tr>
<td>a₃</td>
<td>c₅</td>
<td>d₅</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example

Before B₇

Assume a, b, c, & d defined before B₀
Example

Assume a, b, c, & d defined before B₀

End of B₇

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>5</td>
<td>5</td>
<td>7</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>a₀</td>
<td>b₀</td>
<td>c₀</td>
<td>d₀</td>
<td>i₀</td>
</tr>
<tr>
<td>a₁</td>
<td>b₁</td>
<td>c₁</td>
<td>d₁</td>
<td>i₁</td>
</tr>
<tr>
<td>a₂</td>
<td>b₄</td>
<td>c₂</td>
<td>d₆</td>
<td>i₂</td>
</tr>
<tr>
<td>a₄</td>
<td>c₆</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example

After renaming
- Semi-pruned SSA form
- We’re done …

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

Assume a, b, c, & d defined before B₀
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

\[ i_0 \leftarrow \cdots \]

\[ B_0 \]

\[ i_1 \leftarrow \Phi(i_0, i_2) \]

\[ \cdots \cdots \]

\[ i_2 \leftarrow i_1 + 1 \]

\[ z_0 \leftarrow i_1 + \cdots \]
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