**Static Single Assignment Form**

**Last Time**
- Lattice theoretic frameworks for data-flow analysis

**Today**
- Program representations
- Static single assignment (SSA) form
  - Program representation for sparse data-flow
  - Conversion to and from SSA

**Next Time**
- Reuse optimizations

---

**Data Dependence**

**Definition**
- Data dependences are constraints on the order in which statements may be executed

**Types of dependences**
- **Flow (true) dependence**: $s_1$ writes memory that $s_2$ later reads (RAW)
- **Anti-dependence**: $s_1$ reads memory that $s_2$ later writes (WAR)
- **Output dependences**: $s_1$ writes memory that $s_2$ later writes (WAW)
- **Input dependences**: $s_1$ reads memory that $s_2$ later reads (RAR)

**True dependences**
- Flow dependences represent actual flow of data

**False dependences**
- Anti- and output dependences reflect reuse of memory, not actual data flow; can often be eliminated
Representing Data Dependences

Implicitly
- Use variable defs and uses
- Pros: simple
- Cons: hides data dependence (analyses must find this info)

Def-use chains (du chains)
- Link each def to its uses
- Pros: explicit; therefore fast
- Cons: must be computed and updated, consumes space

Alternate representations
- e.g., Static single assignment form (SSA), dependence flow graphs (DFG), value dependence graphs (VDG)
DU Chains

Definition
– du chains link each def to its uses

Example

\[
\begin{align*}
  s_1 & : a = b; \\
  s_2 & : b = c + d; \\
  s_3 & : e = a + d; \\
  s_4 & : b = 3; \\
  s_5 & : f = b * 2;
\end{align*}
\]

UD Chains

Definition
– ud chains link each use to its defs

Example

\[
\begin{align*}
  s_1 & : a = b; \\
  s_2 & : b = c + d; \\
  s_3 & : e = a + d; \\
  s_4 & : b = 3; \\
  s_5 & : f = b * 2;
\end{align*}
\]
Role of Alternate Program Representations

Process

Original Code (RTL) \[\rightarrow\] SSA Code1 \[\overset{T1}{\rightarrow}\] SSA Code2 \[\overset{T2}{\rightarrow}\] SSA Code3 \[\rightarrow\] Optimized Code (RTL)

Advantage
– Allow analyses and transformations to be simpler & more efficient/effective

Disadvantage
– May not be “executable” (requires extra translations to and from)
– May be expensive (in terms of time or space)

Static Single Assignment (SSA) Form

Idea
– Each variable has only one static definition
– Makes it easier to reason about values instead of variables
– Similar to the notion of functional programming

Transformation to SSA
– Rename each definition
– Rename all uses reached by that assignment

Example
\[\begin{align*}
v & := \ldots \\
\ldots & := \ldots v \\
v & := \ldots \\
\ldots & := \ldots v
\end{align*}\]
\[\begin{align*}
v_0 & := \ldots \\
\ldots & := \ldots v_2 \\
v_1 & := \ldots \\
\ldots & := \ldots v_1
\end{align*}\]

What do we do when there’s control flow?
SSA and Control Flow

Problem
- A use may be reached by several definitions

Merging Definitions
- $\phi$-functions merge multiple reaching definitions

Example
Another Example

 SSA vs. ud/du Chains

 SSA form is more constrained

 Advantages of SSA
 - More compact
 - Some analyses become simpler when each use has only one def
 - Value merging is explicit
 - Easier to update and manipulate?

 Furthermore
 - Eliminates false dependences (simplifying context)

 for (i=0; i<n; i++)
  A[i] = i;

 for (i=0; i<n; i++)
  print(foo(i));

Unrelated uses of i are given different variable names
SSA vs. ud/du Chains (cont)

Worst case du-chains?

```c
switch (c1) {
    case 1:    x = 1; break;
    case 2:    x = 2; break;
    case 3:    x = 3; break;
}
x_4 = \phi(x_1, x_2, x_3)
switch (c2) {
    case 1:    y_1 = x; break;
    case 2:    y_2 = x; break;
    case 3:    y_3 = x; break;
    case 4:    y_4 = x; break;
}
```

m def's and n uses leads to m×n du chains

Transformation to SSA Form

Two steps

- Insert $\phi$-functions
- Rename variables
Where Do We Place $\phi$-Functions?

Basic Rule
- If two distinct (non-null) paths $x \rightarrow z$ and $y \rightarrow z$ converge at node $z$, and
  nodes $x$ and $y$ contain definitions of variable $v$, then a
  $\phi$-function for $v$ is inserted at $z$

\[
\begin{align*}
x & : v_1 := \ldots & y & : v_2 := \ldots \\
x & \downarrow & z & \downarrow \\
v_3 & := \phi(v_1, v_2) & \ldots v_3 \ldots & \\
\end{align*}
\]

Approaches to Placing $\phi$-Functions

Minimal
- As few as possible subject to the basic rule

Briggs-Minimal
- Same as minimal, except $v$ must be live across some edge of the CFG

\[
\begin{align*}
v & = v \\
v & = v \\
\end{align*}
\]

Briggs Minimal will not place a $\phi$ function in this case because $v$ is not live across any CFG edge

no uses of $v$
Approaches to Placing $\phi$-Functions (cont)

**Briggs-Minimal**
- Same as minimal, except $v$ must be live across some edge of the CFG

**Pruned**
- Same as minimal, except dead $\phi$-functions are not inserted

```
\[ v = v \] Will Pruned place a $\phi$ function at this merge?
```

**What’s the difference between Briggs Minimal and Pruned SSA?**

Briggs Minimal will add a $\phi$ function because $v$ is live across the blue edge, but Pruned SSA will not because the $\phi$ function is dead

**Why would we ever use Briggs Minimal instead of Pruned SSA?**
Machinery for Placing $\phi$-Functions

Recall Dominators
- $d \text{ dom } i$ if all paths from entry to node $i$ include $d$
- $d \text{ sdom } i$ if $d \text{ dom } i$ and $d \neq i$

Dominance Frontiers
- The dominance frontier of a node $d$ is the set of nodes that are “just barely” not dominated by $d$; i.e., the set of nodes $n$, such that
  - $d$ dominates a predecessor $p$ of $n$, and
  - $d$ does not strictly dominate $n$
- $DF(d) = \{n | \exists p \in \text{pred}(n), d \text{ dom } p \text{ and } d \not{sdom } n\}$

Notational Convenience
- $DF(S) = \bigcup_{s \in S} DF(s)$

Dominance Frontier Example

$DF(d) = \{n | \exists p \in \text{pred}(n), d \text{ dom } p \text{ and } d \not{sdom } n\}$

Dom(5) = $\{5, 6, 7, 8\}$

DF(5) = $\{4, 5, 12, 13\}$

What’s significant about the Dominance Frontier?
In SSA form, definitions must dominate uses
Dominance Frontier Example II

\[ \text{DF}(d) = \{ n | \exists p \in \text{pred}(n), \ d \text{ dom } p \text{ and } d \not\text{sdom } n \} \]

\[ \text{Dom}(5) = \{5, 6, 7, 8\} \]

\[ \text{DF}(5) = \{4, 5, 13\} \]

In this new graph, node 4 is the first point of convergence between the entry and node 5, so do we need a \( \phi \)-function at node 13?

SSA Exercise

\[ \text{DF}(8) = \{10\} \]
\[ \text{DF}(9) = \{10\} \]
\[ \text{DF}(2) = \{6\} \]
\[ \text{DF}(\{8,9\}) = \{10\} \]
\[ \text{DF}(10) = \{6\} \]
\[ \text{DF}(\{2,6,8,9,10\}) = \{6,10\} \]
**Dominance Frontiers Revisited**

Suppose that node 3 defines variable $x$

$$DF(3) = \{5\}$$

Do we need to insert a $\phi$-function for $x$ anywhere else?

**Yes. At node 6. Why?**

**Dominance Frontiers and SSA**

Let
- $DF_1(S) = DF(S)$
- $DF_{i+1}(S) = DF(S \cup DF_i(S))$

**Iterated Dominance Frontier**
- $DF_\infty(S)$

**Theorem**
- If $S$ is the set of CFG nodes that define variable $v$, then $DF_\infty(S)$ is the set of nodes that require $\phi$-functions for $v$
Algorithm for Inserting $\phi$-Functions

for each variable $v$
    WorkList $\leftarrow$ $\emptyset$
    EverOnWorkList $\leftarrow$ $\emptyset$
    AlreadyHasPhiFunc $\leftarrow$ $\emptyset$
    for each node $n$ containing an assignment to $v$
        WorkList $\leftarrow$ WorkList $\cup$ $\{n\}$
        EverOnWorkList $\leftarrow$ WorkList
    while WorkList $\neq$ $\emptyset$
        Remove some node $n$ from WorkList
        for each $d \in$ DF($n$)
            if $d \notin$ AlreadyHasPhiFunc
                Insert a $\phi$-function for $v$ at $d$
                AlreadyHasPhiFunc $\leftarrow$ AlreadyHasPhiFunc $\cup$ $\{d\}$
            if $d \notin$ EverOnWorkList
                Process each node at most once
                WorkList $\leftarrow$ WorkList $\cup$ $\{d\}$
                EverOnWorkList $\leftarrow$ EverOnWorkList $\cup$ $\{d\}$

Put all defs of $v$ on the worklist
Insert at most one $\phi$ function per node
Process each node at most once

Variable Renaming

Basic idea
- When we see a variable on the LHS, create a new name for it
- When we see a variable on the RHS, use appropriate subscript

Easy for straightline code

Use a stack when there’s control flow
- For each use of $x$, find the definition of $x$ that dominates it

Traverse the dominance tree
Dominance Tree Example

The dominance tree shows the dominance relation

Variable Renaming (cont)

Data Structures
- Stacks[v] ∀v
  Holds the subscript of most recent definition of variable v, initially empty
- Counters[v] ∀v
  Holds the current number of assignments to variable v; initially 0

Auxiliary Routine
procedure GenName(variable v)
i := Counters[v]
push i onto Stacks[v]
Counters[v] := i + 1

Use the Dominance Tree to remember the most recent definition of each variable
Variable Renaming Algorithm

\begin{verbatim}
procedure Rename(block b)
    for each \( \phi \)-function \( p \) in \( b \)
        GenName(LHS(p)) and replace \( v \) with \( v_i \), where \( i = \text{Top(Stack}[v]) \)
    for each statement \( s \) in \( b \) (in order)
        for each variable \( v \) \( \in \) RHS(s)
            replace \( v \) by \( v_i \), where \( i = \text{Top(Stacks}[v]) \)
        for each variable \( v \) \( \in \) LHS(s)
            GenName(v) and replace \( v \) with \( v_i \), where \( i = \text{Top(Stack}[v]) \)
    for each \( s \) \( \in \) succ(b) (in CFG)
        \( j \leftarrow \) position in \( s \)'s \( \phi \)-function corresponding to block \( b \)
        for each \( \phi \)-function \( p \) in \( s \)
            replace the \( j \)th operand of RHS(p) by \( v_i \), where \( i = \text{Top(Stack}[v]) \)
    for each \( s \) \( \in \) child(b) (in DT)
        Rename(s)
        for each \( \phi \)-function or statement \( t \) in \( b \)
            for each \( v_i \) \( \in \) LHS(t)
                Pop(Stack[v])
        \} \text{ Unwind stack when done with this node}
    \} \text{ Recurse using Depth First Search}
\end{verbatim}

Transformation from SSA Form

Proposal
- Restore original variable names (\textit{i.e.}, drop subscripts)
- Delete all \( \phi \)-functions

Complications
- What if versions get out of order? (simultaneously live ranges)

Alternative
- Perform dead code elimination (to prune \( \phi \)-functions)
- Replace \( \phi \)-functions with copies in predecessors
- Rely on register allocation coalescing to remove unnecessary copies
Concepts

Data dependences
- Three kinds of data dependences
- du-chains

Alternate representations

SSA form

Conversion to SSA form
- $\phi$-function placement
  - Dominance frontiers
- Variable renaming
  - Dominance trees

Conversion from SSA form

Next Time

Assignments
- Project proposals due

Lecture
- Reuse optimizations