Control-Flow Analysis and Loop Detection

Last time
- Lattice-theoretic framework for data-flow analysis

Today
- Control-flow analysis
- Loops
- Identifying loops using dominators
- Converting to SSA using dominators
- Dominators and PA2
Context

Data-flow
- Flow of data values from defs to uses
- Could alternatively be represented as a data dependence

Control-flow
- Sequencing of operations
- Could alternatively be represented as a control dependence
- e.g., Evaluation of then-code and else-code depends on if-test

Why study control flow analysis?

Finding Loops
- most computation time is spent in loops
- to optimize them, we need to find them

Loop Optimizations
- Loop-invariant code hoisting
- Induction variable elimination
- Array bounds check removal
- Loop unrolling
- Parallelization
- ...

Identifying structured control flow
- can be used to speed up data-flow analysis
Representing Control-Flow

High-level representation
– Control flow is implicit in an AST

Low-level representation:
– Use a Control-flow graph
  – Nodes represent statements
  – Edges represent explicit flow of control

Other options
– Control dependences in program dependence graph (PDG) [Ferrante87]
– Dependences on explicit state in value dependence graph (VDG) [Weise 94]

What Is Control-Flow Analysis?

Control-flow analysis discovers the flow of control within a procedure
(e.g., builds a CFG, identifies loops)

Example

```
1  a := 0
2  b := a * b
3  L1:  c := b/d
4  if c < x goto L2
5  e := b / c
6  f := e + 1
7  L2:  g := f
8  h := t - g
9  if e > 0 goto L3
10 goto L1
11 L3:  return
```

Diagram:

```
  a := 0
  b := a * b
  c := b/d
  c < x?
      e := b / c
      f := e + 1
  L2:  g := f
  h := t - g
  if e > 0 goto L3
  goto L1
  L3:  return
```

Diagram:

```
  a := 0
  b := a * b
  c := b/d
  c < x?
      e := b / c
      f := e + 1
  L2:  g := f
  h := t - g
  e > 0?
      No
      Yes
  goto L1
  L3:  return
```
**Loop Concepts**

**Loop**: Strongly connected subgraph of CFG with a single entry point (header)

**Loop entry edge**: Source not in loop & target in loop

**Loop exit edge**: Source in loop & target not in loop

**Loop header node**: Target of loop entry edge. Dominates all nodes in loop.

**Back edge**: Target is loop header & source is in the loop

**Natural loop**: Associated with each back edge. Nodes dominated by header and with path to back edge without going through header

**Loop tail node**: Source of back edge

**Loop preheader node**: Single node that’s source of the loop entry edge

**Nested loop**: Loop whose header is inside another loop

---

**Picturing Loop Terminology**

![Diagram of loop terminology](image-url)
The Value of Preheader Nodes

Not all loops have preheaders
   – Sometimes it is useful to create them

Without preheader node
   – There can be multiple entry edges

With single preheader node
   – There is only one entry edge

Useful when moving code outside the loop
   – Don’t have to replicate code for multiple entry edges

Identifying Loops

Why?
   – Most execution time spent in loops, so optimizing loops will often give most benefit

Many approaches
   – Interval analysis
      – Exploit the natural hierarchical structure of programs
      – Decompose the program into nested regions called intervals
   – Structural analysis: a generalization of interval analysis
   – Identify dominators to discover loops

We’ll focus on the dominator-based approach
### Dominator Terminology

**Dominator Terminology**

- **Dominator**
  
  \[ d \text{ dom} \ i \ ⇔ \ \text{if all paths from entry to node } i \ \text{include} \ d \]

- **Strict dominators**
  
  \[ d \text{ sdom} \ i \ ⇔ \ d \text{ dom} \ i \ \text{and} \ d \neq i \]

- **Immediate dominators**
  
  \[ a \text{ idom} \ b \ ⇔ \ a \text{ sdom} \ b \ \text{and there does not exist} \ c \neq a, c \neq b, a \text{ dom} \ c, \ \text{and} \ c \text{ dom} \ b \]

- **Post dominators**
  
  \[ p \text{ pdom} \ i \ ⇔ \ \text{if every possible path from } i \ \text{to exit includes} \ p \ (p \text{ dom} \ i \ \text{in the flow graph whose arcs are reversed and entry and exit are interchanged}) \]

#### Identifying Natural Loops with Dominators

**Back edges**

A *back edge* of a natural loop is one whose target dominates its source.

**Natural loop**

The *natural loop* of a back edge \((m \rightarrow n)\), where \(n\) dominates \(m\), is the set of nodes \(x\) such that \(n\) dominates \(x\) and there is a path from \(x\) to \(m\) not containing \(n\).

**Example**

SCC with \(c\) and \(d\) not a loop because has two entry points. The target, \(c\), of the edge \((d \rightarrow c)\) does not dominate its source, \(d\), so \((d \rightarrow c)\) does not define a natural loop.
Computing Dominators

**Input:** Set of nodes $N$ (in CFG) and an entry node $s$

**Output:** $\text{Dom}[i] = \text{set of all nodes that dominate node } i$

\[
\begin{align*}
\text{Dom}[s] &= \{s\} \\
\text{for each } n &\in N - \{s\} \\
\text{Dom}[n] &= N \\
\text{repeat} \\
\text{change} &= \text{false} \\
\text{for each } n &\in N - \{s\} \\
D &= \{n\} \cup (\cap_{p \in \text{pred}(n)} \text{Dom}[p]) \\
\text{if } D &\neq \text{Dom}[n] \\
\text{change} &= \text{true} \\
\text{Dom}[n] &= D \\
\text{until} \ \text{!change}
\end{align*}
\]

**Key Idea**

If a node dominates all predecessors of node $n$, then it also dominates node $n$.

$x \in \text{Dom}(p_1) \land x \in \text{Dom}(p_2) \land x \in \text{Dom}(p_3) \Rightarrow x \in \text{Dom}(n)$

Computing Dominators (example)

**Input:** Set of nodes $N$ and an entry node $s$

**Output:** $\text{Dom}[i] = \text{set of all nodes that dominate node } i$

\[
\begin{align*}
\text{Dom}[s] &= \{s\} \\
\text{for each } n &\in N - \{s\} \\
\text{Dom}[n] &= N \\
\text{repeat} \\
\text{change} &= \text{false} \\
\text{for each } n &\in N - \{s\} \\
D &= \{n\} \cup (\cap_{p \in \text{pred}(n)} \text{Dom}[p]) \\
\text{if } D &\neq \text{Dom}[n] \\
\text{change} &= \text{true} \\
\text{Dom}[n] &= D \\
\text{until} \ \text{!change}
\end{align*}
\]

Initially

$\text{Dom}[s] = \{s\}$

$\text{Dom}[q] = \{n, p, q, r, s\} \ldots$

Finally

$\text{Dom}[q] = \{q, s\}$

$\text{Dom}[r] = \{r, s\}$

$\text{Dom}[p] = \{p, s\}$

$\text{Dom}[n] = \{n, p, s\}$
Recall SSA, Another use of dominator information

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)

Process

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

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
```
v := \ldots
\ldots := \ldots v \ldots
v := \ldots
\ldots := \ldots v \ldots
```
```
v_0 := \ldots
\ldots := \ldots v_0 \ldots
v_1 := \ldots
\ldots := \ldots v_1 \ldots
```

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

Problem
- A use may be reached by several definitions

\[
\begin{align*}
1 & & \\
2 & v := \ldots & 3 & v := \ldots \\
4 & \ldots v \ldots & & \end{align*}
\]

Merging Definitions
- \( \phi \)-functions merge multiple reaching definitions

\[
\begin{align*}
1 & & \\
2 & v_0 := \ldots & 3 & v_1 := \ldots \\
4 & \ldots v \ldots & & \\
2 & v_2 := \phi(v_0, v_1) & & \\
& \ldots v_2 \ldots & &
\end{align*}
\]
Another Example

Transformation to SSA Form

Two steps
- Insert $\phi$-functions
- Rename variables
Where Do We Place φ-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 φ-function for \( v \) is inserted at \( z \)

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

Machinery for Placing φ-Functions

Recall Dominators
- \( d \) dom \( i \) if all paths from entry to node \( i \) include \( d \)
- \( d \) sdom \( i \) if \( d \) 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 \( n \)
- \( DF(d) = \{ n | \exists p \in \text{pred}(n), \text{d dom } p \text{ and } d \text{ !sdom } n \} \)

Notational Convenience
- \( DF(S) = \bigcup_{n \in S} DF(n) \)
**Dominance Frontier Example**

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

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

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

In SSA form, definitions must dominate uses

**Dominance Frontier Example II**

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

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

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

In this 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

<table>
<thead>
<tr>
<th>Node</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>(v_3 := \ldots)</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>(v_5 := \phi(v_3, v_4))</td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>(v_1 := \ldots)</td>
</tr>
<tr>
<td>9</td>
<td>(v_2 := \ldots)</td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

DF(8) = \{10\}
DF(9) = \{10\}
DF(2) = \{6\}  \(DF(d) = \{n \mid \exists p \in \text{pred}(n), d \text{ dom } p \text{ and } d \not\text{ sdom } n\}\)
DF({8,9}) = \{10\}
DF(10) = \{6\}
DF({2,8,9,6,10}) = \{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 \(v\)

Dominance Tree Example

*The dominance tree shows the dominance relation*

![Dominance Tree](image)
Inserting Phi Nodes

Calculate the dominator tree
- a lot of research has gone into calculating this quickly

Computing dominance frontier from dominator tree
- $DF_{\text{local}}[n] =$ successors of $n$ (in CFG) that are not strictly dominated by $n$
- $DF_{\text{up}}[n] =$ nodes in the dominance frontier of $n$ that are not strictly dominated by $n$’s immediate dominator
- $DF[n] = DF_{\text{local}}[n] \cup \bigcup_{c \in \text{children}[n]} DF_{\text{up}}[c]$

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 $v$  \hspace{1cm} \textit{Put all defs of $v$ on the worklist}
    WorkList $\leftarrow$ WorkList $\cup \{n\}$
    EverOnWorkList $\leftarrow$ WorkList
  while WorkList $\neq \emptyset$
    Remove some node $n$ for WorkList
    for each $d \in DF(n)$
      if $d \notin$ AlreadyHasPhiFunc  \hspace{1cm} \textit{Insert at most one $\phi$ function per node}
        Insert a $\phi$-function for $v$ at $d$
        AlreadyHasPhiFunc $\leftarrow$ AlreadyHasPhiFunc $\cup \{d\}$
      if $d \notin$ EverOnWorkList  \hspace{1cm} \textit{Process each node at most once}
        WorkList $\leftarrow$ WorkList $\cup \{d\}$
        EverOnWorkList $\leftarrow$ EverOnWorkList $\cup \{d\}$
Transformation to SSA Form

Two steps
- Insert $\phi$-functions
- Rename variables

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*

```
  x = x
  x = x

  x₀ = x₀
  x₁ = x₁
```

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

```
  x = x

  x₀ = x₀
```

Traverse the dominance tree
Variable Renaming (cont)

Data Structures
- \( \text{Stacks}[v] \forall v \)
  Holds the subscript of most recent definition of variable \( v \), initially empty
- \( \text{Counters}[v] \forall v \)
  Holds the current number of assignments to variable \( v \); initially 0

Auxiliary Routine

\[
\text{procedure } \text{GenName}(\text{variable } v) \\
i := \text{Counters}[v] \\
\text{push } i \text{ onto } \text{Stacks}[v] \\
\text{Counters}[v] := i + 1
\]

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

Variable Renaming Algorithm

\[
\text{procedure } \text{Rename}(\text{block } b) \\
\text{if } b \text{ previously visited return } \\
\text{Call Rename(entry-node)}
\]

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

\[
\text{Unwind stack when done with this node}
\]
Transformation from SSA Form

Proposal
- Restore original variable names (i.e., drop subscripts)
- Delete all φ-functions

Complications (the proposal doesn’t work!)
- What if versions get out of order? (simultaneously live ranges)

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

PA2 and Dominators

Why might you be getting ‘Instruction does not dominate all uses!’ error?
Next Time

Reading
– Advanced Compiler Optimizations for Supercomputers by Padua and Wolfe

Lecture
– Dependencies in loops
– Parallelization and Performance Optimization of Applications