Control Flow Analysis

CS2210
Lecture 11

Reading & Topics

- Muchnick: chapter 7
- Optimization Overview
- Control Flow Analysis
- Maybe start data flow analysis

Optimization Overview

- Two step process
  - Analyze program to learn things about it “program analysis”
  - Determine when transformations are legal & profitable
  - Transform the program based on information into semantically equivalent but better output program
- Optimization is a misnomer
  - Almost never optimal
  - Sometimes slows some programs down on some inputs (try to speed up most programs on most inputs)
Semantics

- Subtleties
  - Evaluation order
  - Arithmetic properties (e.g. associativity)
  - Behavior in error cases
- Some languages very precise
  - E.g., Ada
- Some weaker
  - Potentially more optimization opportunity

Analysis Scope

- Peephole
  - Across small number of adjacent instructions
  - Trivial
  - Local
    - Within a basic block
    - Simple
- Intraprocedural (aka. Global)
  - Across basic blocks within a procedure
  - More complex, branches, merges loops
- Interprocedural
  - Across procedures, within whole program
  - More complex, calls, returns
  - More useful for higher-level languages
  - Hard with separate compilation
  - Whole-program
    - Useful for safety properties
    - Most complex

Catalog of Optimizations

- Arithmetic simplification
  - Constant folding
    - \( x := 3+4 \Rightarrow x := 7 \)
  - Strength reduction
    - \( x := y^4 \Rightarrow x := y < 2 \)
- Constant propagation
  - \( x := 5 \Rightarrow x := 5 \)
  - \( y := x+2 \Rightarrow y := 5+2 \)
  - \( y := 7 \)

- Copy propagation
  - \( x := y \Rightarrow x := y \)
  - \( w := w + x \Rightarrow w := w + y \)
Catalog (2)

- Common Subexpression Elimination (CSE)
  \[ x := a + b \Rightarrow x := a + b \]
  ...  \[ y := a + b \Rightarrow y := x \]

- Can also eliminate redundant memory references, branch tests

- Partial Redundancy Elimination (PRE)
  Like CSE but earlier expression available only along some path

  \[
  \begin{align*}
  x & := a + b & t & := a + b; x := t \\
  \text{if} \quad \text{then} & \quad \text{if} \quad \text{then} \\
  y & := a + b & x & := a + b \quad \text{end} & \quad \text{else} & \quad \text{end} \\
  \quad \ldots & \quad \ldots & \quad \ldots & \quad \ldots \\
  y & := a + b & y & := t
  \end{align*}
  \]

Catalog (3)

- Pointer analysis
  \[ p := \& x \Rightarrow p := \& x \quad ^{*p} := 5 \quad ^{*p} := 5 \]
  \[ y := x + 1 \quad y := 6 \]

- Dead assignment elimination
  \[ x := y + z \]
  ... /* no use of x */
  \[ x := 6 \]

- Dead code elimination
  \[ \text{if} \quad \text{false} \quad \text{then} \]

- Integer range analysis
  \[
  \begin{align*}
  \text{for} \quad (i && 0; i < 10; i++) \quad \{ \\
  a[i] & := 0; \\
  \}
  \end{align*}
  \]

Loop Optimizations (1)

- Loop-invariant code motion
  \[
  \begin{align*}
  \text{for} \quad j & := 1 \quad \text{to} \quad 10 \\
  \quad \text{for} \quad i & := 1 \quad \text{to} \quad 10 \\
  \quad \quad a[i] & := a[i] + b[j] \\
  \}
  \quad \text{for} \quad j & := 1 \quad \text{to} \quad 10 \\
  \quad \quad t & := b[j] \\
  \quad \text{for} \quad i & := 1 \quad \text{to} \quad 10 \\
  \quad \quad a[i] & := a[i] + t
  \end{align*}
  \]

- Induction variable elimination
  \[
  \begin{align*}
  \text{for} \quad i & := 1 \quad \text{to} \quad 10 \\
  a[i] & := a[i] + 1 \\
  \}
  \quad \text{for} \quad p & := \& a[1] \quad \text{to} \quad \& a[10] \\
  \quad \quad ^{*p} & := ^{*p} + 1 \\
  \quad \quad a[i] \quad \text{is several} \\
  \quad \quad \text{instructions} \quad ^{*p} \quad \text{is one}
  \end{align*}
  \]
Loop Optimizations (2)

- Loop unrolling
  for i := 1 to N
  a[i] := a[i] + 1
→ for i := 1 to N by 4
  a[i] := a[i] + 1
  a[i+1] := a[i+1] + 1
  a[i+2] := a[i+2] + 1
  a[i+3] := a[i+3] + 1
- Creates more optimization opportunities in loop body

- Parallelization
- Interchange
- Reversal
- Fusion
- Blocking / tiling
  - Data cache locality optimization

Call Optimizations

- Inlining
  l := ...
  w := 4
  a := area(l,w)
→ l := ...
  w := 4
  a := l*w
⇒ l := ...
  w := 4
  a := l << 2

- Many simple optimizations become important after inlining
- Interprocedural constant propagation

More Call Optimizations

- Static binding of dynamic calls
  - Calls through function pointers in imperative languages
  - Call of computed function in functional language
  - OO-dispatch in OO languages (e.g., COOL)
    - If receiver class can be deduced, can replace with direct call
  - Other optimizations possible even when multiple targets
    (e.g., using PICs = polymorphic inline caches)
- Procedure specialization
  - Partial evaluation
Machine-dependent Optimizations

- Register allocation
- Instruction selection
  - Important for CISCs
- Instruction scheduling
  - Particularly important with long-delay instructions and on wide-issue machines (superscalar + VLIW)

The Phase Ordering Problem

- In what order should optimizations be performed?
  - Some optimizations create opportunities for others (order according to this dependence)
  - Can make some optimizations simple
    - Later optimization will “clean up”
    - What about adverse interactions
      - Common subexpression elimination ⇔ register allocation
      - Register allocation ⇔ instructions scheduling
    - What about cyclic dependences?
      - Constant folding ⇔ constant propagation

Control Flow Analysis
Approaches

- **Dominator-based**
  - Control flow graph with dominator relation to identify loops
  - Most commonly used

- **Interval-based**
  - Nested regions (= intervals)
  - Control tree
  - Special case: structural analysis
    - Most sophisticated
    - Classifies control structures (not just loops)

Basic Blocks and Control Flow Graphs (CFGs)

- **Basic block** = maximal sequence of instructions entered only from first and exited from last
- Entry can be
  - Procedure entry point
  - Branch target
  - Instruction immediately following branch or return
- Entry instructions are called **leaders**

Example

```
receive m
f0 <- 0
f1 <- 1
if m <= 1 goto L3
i <- 2
L1: if i <= m goto L2
    return r
    f2 <- f0 + f1
    f0 <- f1
    f1 <- f2
    i <- i + 1
    L3: return m
```

```
Example CFG

Dominators & Postdominators
- Binary relation useful to determine loops
- Node $d \text{dom} i$ $\iff$ every possible execution path from entry to $i$ includes $d$
- Dominance relation is
  - Reflexive: $d \text{dom} d$
  - Transitive: $a \text{dom} b$ and $b \text{dom} c$ then $a \text{dom} c$
  - Anti-symmetric: if $a \text{dom} b$ and $b \text{dom} a$ then $a = b$
- Immediate dominance
  - For $a = b$: $a \text{idom} b$ iff $a \text{dom} b$ and $\forall c: c \text{dom} b$ and $a \text{dom} c \implies c = a$

Dominators & Postdominators
- $p \text{pdom} i$ $\iff$ every possible execution path from node $i$ to exit includes $p$
- Dual relation: $i \text{dom} p$ in CFG with edges reversed and entry and exit switched
- $a \text{ strictly dominates} b$ $\iff$ $a \text{dom} b$ and $a \neq b$
Computing Dominators (easy but slow)
- Initialize $\text{dom}(i) = \text{set of all nodes for } i = \text{entry}$, $\text{dom}(\text{entry}) = \{\text{entry}\}$
- While changes occur do
  - $\text{dom}(i) = \{i\} \cup \text{dom}(\text{pred}(i))$ for all predecessors of $i$
- Works fastest if nodes are processed in reverse postorder
  - $O(n^2e)$ complexity, $n$ number of nodes, $e$ number of edges

Another Example
- Preorder: 0 - 1 - 2 - 3 - 4 - 5 - 6
- Postorder: 5 - 4 - 3 - 6 - 2 - 1 - 0
- Reverse Postorder: 0 - 1 - 2 - 6 - 3 - 4 - 5
**Computing IDOM**
- Compute dominators
  - $tmp(i) := dom(i) - \{i\}$
  - Remove all $n \in tmp(i)$ from $tmp(i)$ for which $\exists x (\neq n) \in tmp(i)$ such that $n \in tmp(x)$

**Computing Dominators Faster**
- Lengauer & Tarjan’s algorithm
  - Described in the book
  - $O(e^{\alpha(e,n)})$ running time where $\alpha$ is the inverse of Ackerman’s function
Loops & SCCs

- An edge \( e = (m, n) \) is called a **back edge** iff \( n \) dom \( m \) (head dominates tail).
- A **natural loop** of back edge \( m \rightarrow n \) := subgraph containing \( n \) and all nodes from which \( m \) can be reached w/o passing through \( n \) and the edges that connect those nodes.
  - \( n \) is called the **loop header**.
  - **Preheader** a node inserted immediately before the loop header.
  - Useful in many loop optimization as a "landing pad" for code from the loop body.

Headers and Preheaders

- ![Diagram of headers and preheaders](image)

Natural Loop Properties

- Natural loops with different headers are either
  - Nested
  - Disjoint
- What about natural loops with the same header?

![Diagram of natural loop properties](image)
Strongly Connected Components

- Generalization of loops
- A SCC is a subgraph $G_S = <N_S, E_S>$ in which every node in $N_S$ is reachable from every other node in $N_S$ by a path including only edges from $E_S$
- An SCC $S$ is **maximal**, iff every SCC containing it is the $S$ itself

Reducible Flow Graphs

- A flow graph $G=(N,E)$ is **reducible** (aka. well-structured) if $E$ can be partitioned into $E = E_B \cup E_F$, where $E_B$ is the back edge set, so that $(N, E_F)$ forms a DAG in which all nodes are reachable from the entry node
- Patterns that make CFGs irreducible, are called **improper regions**
  - Impossible in some languages (e.g., Modula-2)

Dealing with Irreducibility

- Cannot use structural analysis directly
- Use **iterative data flow** analysis instead
- Make graph well-structured using **node splitting**
  - **Induced iteration** on the lattice of monotone functions from the lattice to itself (more on this later)
Control Trees & Interval Analysis

- Idea:
  - divide CFG into regions of various kinds
  - Combine each region into a new node (abstract node)
  - Obtain an abstract flow graph
  - Final result is called control tree
  - Root of control tree is abstract flow graph representing original flow graph
  - Leaves of control tree are basic blocks
  - Nodes between root and leaves represent regions
  - Edges represent relationships between abstract node and its descendents

Example: T1-T2 Analysis

Interval Analysis

- A (maximal) interval $I_m(h)$ with header $h$ is the maximal single-entry subgraph with $h$ as only entry node and with all closed subpaths in the subgraph containing $h$
  - Like natural loop but with “acyclic stuff dangling off loop exits”
- A (minimal) interval is either
  - A natural loop
  - A maximal acyclic subgraph
  - A minimal irreducible region
Interval Analysis Steps

- Iterate until done:
  - Postorder traversal of CFG looking for loop headers
  - Construct natural loop for each loop header and reduce the loop to an abstract region “natural loop”
  - For each set of entries of an improper region construct minimal SCC and reduce it to “improper region”
  - For entry node and each immediate descendant of a node in a natural loop or irreducible region construct maximal acyclic graph with that node as root: if more than one node results, reduce to “acyclic region”

Structural Analysis

- A refinement of interval analysis
- Advantage compared to standard iterative data flow analysis
  - Uses specialized flow functions for recognized structures that are much faster
  - Data flow equations are determined by the syntax and semantics of the (source) language
  - Recognizes more structures than standard interval analysis

Region Types

- Blocks
- If-then
- If-then-else
- Case-switch
- Self loop
- While loop
- Natural loop
- Improper interval
- Proper interval
Data Flow Analysis

Reading
- Muchnick Chapter 8
- Some material not in book just in lecture notes

Data Flow Analysis
- Compute information about program
  - At program points
  - To identify optimization opportunities
- Can model as solving a system of constraints
  - Each CFG node imposes a constraint relating predecessor and successor info
  - Solution to constraints is result of analysis
  - Solution must be
    - Sound (aka safe)
    - Can be conservative
Key Issues

- Do constraints define analysis correctly?
- How to efficiently represent information?
- How to represent and solve constraints efficiently?
- What about multiple solutions?
- How synchronize transformation with analysis?

Example: Reaching Definitions

- For each program point want to know
  - What set of definitions (statements) may reach that point
  - Reach = the last definition of a variable
  - Info = set of var -> LIR bindings, e.g., \{x->s_1, y->s_2, y->s_3\}
  - Can use the info to
    - Build def-use chains
    - Do constant- and copy propagation
  - Safety
    - Can have more bindings that "true" answer

Constraints for Reaching Defs

- **Strong update**
  
  \[ s' \triangleq x := \ldots \]
  
  info_{\text{succ}} = info_{\text{pred}} - (x->s') \cup (x->s)

- **Weak update**
  
  \[ s' \triangleq *p := \ldots \]
  
  info_{\text{succ}} = info_{\text{pred}} \cup (x->s) \forall x \in \text{may-point-to}(p)

- Other statements do nothing
  
  info_{\text{succ}} = info_{\text{pred}}
More Constraints for RDEFs

- Branches
  \[ \text{info}_{\text{pred}}(i) = \text{info}_{\text{succ}}(i) \]

- Control flow merges
  - We don’t know which path is taken at run time
  - Be conservative:
    \[ \text{info}_{\text{pred}} = \bigcup_i \text{info}_{\text{pred}(i)} \]

- Procedure entry:
  - receive x
    \[ \text{info} = \{ x \rightarrow \text{entry} \} \]

Solving Constraints

- For reaching definitions can solve traversing instructions in forward topological order

\[
\begin{align*}
(1) & \ x := -_\rightarrow \\
(2) & \ y := -_\rightarrow \\
(3) & \ y := -_\rightarrow \\
(4) & \ p := -_\rightarrow \\
(5) & \ x := -_\rightarrow \\
(6) & \ x := -_\rightarrow \\
(7) & \ *p := -_\rightarrow \\
(8) & \ y := -_\rightarrow \\
\end{align*}
\]

Another Example

\[
\begin{align*}
(1) & \ x := -_\rightarrow \\
(2) & \ y := -_\rightarrow \\
(3) & \ y := -_\rightarrow \\
(4) & \ p := -_\rightarrow \\
(5) & \ x := -_\rightarrow \\
(6) & \ x := -_\rightarrow \\
(7) & \ *p := -_\rightarrow \\
(8) & \ y := -_\rightarrow \\
\end{align*}
\]
Constraints and Loops

- Constraints are now recursively defined
  - $\text{info}_{\text{loop-head}} = \text{info}_{\text{loop-entry}} \cup \text{info}_{\text{back-edge}}$
  - $\text{info}_{\text{back-edge}} = \ldots \text{info}_{\text{loop-head}} \ldots$
- Substituting definition of $\text{info}_{\text{back-edge}}$:
  - $\text{info}_{\text{loop-head}} = \text{info}_{\text{loop-entry}} \cup (\ldots \text{info}_{\text{loop-head}} \ldots)$
  - Summarized rhs as $F$:
  - $\text{info}_{\text{loop-head}} = F(\text{info}_{\text{loop-head}})$
- Legal solutions to constraints is a fixed-point of $F$

Recursive Constraints

- Many solutions possible
  - Want least or greatest fixed-point, whichever corresponds to most precise answer
- How can fixed-point be found
  - Interval & structural analysis for certain CFGs
  - Iterative approximation for arbitrary CFGs
Iterative Data Flow Analysis

- Start with initial guess of info at loop head
  \[ \text{info}_{\text{loop head}} = \text{guess} \]
- Solve equations for body
  \[ \text{info}_{\text{back-edge}} = F_{\text{body}}(\text{info}_{\text{loop head}}) \]
  \[ \text{info}_{\text{loop head}} = \text{info}_{\text{loop entry}} \cup \text{info}_{\text{back-edge}} \]
- Test if fixed-point found
  \[ \text{info}_{\text{loop head}} = \text{info}_{\text{loop head}} \]
- If same done
- Else use result as new (better) guess:
  \[ \text{info}_{\text{back-edge}} = F_{\text{body}}(\text{info}_{\text{loop head}}) \]
  \[ \text{info}_{\text{loop head}} = \text{info}_{\text{loop entry}} \cup \text{info}_{\text{back-edge}} \]

When does Iterating work?

- Have to be able to make initial guess
- Info^{n+1} must be closer to fixed-point than info^n
- Must eventually reach fixed-point, info must be drawn from a finite height domain
- To reach best fixed-point, initial guess should be optimistic
  \[ \text{info}_{\text{loop head}} = \text{info}_{\text{loop entry}} \]
- Even if guess is overly optimistic, iteration will ensure that we get a safe
- Iteration speed
  - Ideal: solve constraints along shortest path from loop head to tail
  - Practical: avoid solving constraints outside of loop until fixed-point reached inside loop

Data Flow Analysis Direction

- Constraints are declarative, so may require mix of forward and backward
- Frequently directional propagation & iteration
  - Forward or backward
  - Topological traversal of acyclic subgraph
  - Minimizes analysis time
- Directional constraints are called flow functions
  \[ R_{\text{def}, x \rightarrow s} : \text{in} = \text{in} - \{x \rightarrow s\} \cup \{x \rightarrow s\} \]
**GEN & KILL sets**

- Flow functions can often be described by their GEN and KILL sets
- GEN = new information added
- KILL = old information removed
- \( F_{\text{out}}(m) = \text{in} \setminus \text{KILL}_{\text{out}} \cup \text{GEN}_{\text{out}} \)
- Example reaching definitions
  - \( \text{GEN}_{x \leftarrow ...} = \{ x \rightarrow s \} \)
  - \( \text{KILL}_{x \leftarrow ...} = \{ x \rightarrow s | \forall x' \} \)

**Bit Vectors**

- Encode data flow information and GEN/KILL sets as bit vectors
- Works when info can be expressed abstractly as a set of things
- Each gets a specific bit position
- Reaching defs: info = bit vector over statements, each bit represents a specific statement, defined variable is implied by statement
- Advantages
  - Compact representation
  - Fast union & difference operations
  - Can combine GEN / KILL sets of whole basic block into one GEN/KILL set for faster iteration

**Example: Constant Propagation**

- What info should be represented?
- \( CP_{\text{const}}(m) = \)
- \( CP_{x := y \times z}(m) = \)
- Merge function?
- Initial info?
- Direction?
- Can bit vectors be used?
Another Example: Constant Propagation

\[ x := 3 \]
\[ x := x + 1 \]
\[ w := 3 \]
\[ w := 3 \]
\[ y := x \times 2 \]
\[ z := y + 5 \]
\[ w := w \times 2 \]

May vs. Must Info

- Some kind of info implies guarantees: **must info**
- Some kind of info implies possibilities: **may info**

<table>
<thead>
<tr>
<th>Desired info</th>
<th>May</th>
<th>Must</th>
</tr>
</thead>
<tbody>
<tr>
<td>Safe</td>
<td>Overly big set</td>
<td>Overly small set</td>
</tr>
<tr>
<td>GEN</td>
<td>Add everything that might be true</td>
<td>Add only if guaranteed true</td>
</tr>
<tr>
<td>KILL</td>
<td>Remove only if guaranteed true</td>
<td>Remove everything possibly wrong</td>
</tr>
<tr>
<td>MERGE</td>
<td>Union</td>
<td>Union</td>
</tr>
</tbody>
</table>

Live Variables Analysis

- Desired info: set of variables **live** at each program point
  - Live = might be used later in the program
  - Supports dead assignment elimination, register allocation
- May or must?
- Direction?
- Merge function
- Bit vectors usable?
- Initial info?
Live Variables Example

\[
\begin{align*}
x &:= 5 \\
y &:= x^2 \\
x &:= x + 1 \\
y &:= x + 10
\end{align*}
\]

Lattice-theoretic Data Flow Analysis Framework

- **Goals**
  - Provide single, formal model to describe all DFAs
  - Formalize "safe", "conservative", "optimistic"
  - Precise bounds on time complexity
  - Connect analysis with underlying semantics to enable correctness proofs

- **Define domain of program properties computed by DFA**
  - Each domain has set of elements
  - Each element represents one possible value of the property
  - Order sets to reflect relative precision
  - Domain = set of elements + order over elements = **lattice**

- **Plan**
  - Define flow/merge functions using lattice operators
  - Benefit from lattice theory for realizing goals

Lattices

- **D = (S, \leq)**
  - S set of elements
  - \(\leq\) induces a **partial order**
  - Reflexive, transitive & anti-symmetric
  - \(\forall x, y \in S: \text{meet } (x, y)\) (greatest lower bound)
  - \(\text{join } v(x, y)\) (least upper bound)
  - \(\nu\) = closure property

- **Unique Top (T) & Bottom (⊥) elements:**
  - \(\forall x \in S: x \land T = T\)
  - Meet and join are commutative and associative

- **Height of lattice:** longest path through partial order from top to bottom
Lattices in Data Flow Analysis

- Model information by elements of a lattice domain
  - Top = best case info
  - Bottom = worst case info
  - Initial info for optimistic analyses (at least back edges: top)
  - If $a \leq b$ then $a$ is a conservative approximation of $b$
  - Merge function = meet ($\land$): the most precise element that's a conservative approximation of both input elements
  - Initial info for optimistic analyses (at least back edges: top)

Some Typical Lattice Domains

- Two point lattice: $\perp, T$
  - Boolean property
  - A tuple of two point lattices = bit vector
  - Lifted set: set of incomparable values and $\perp$ and $T$
    - Example?
    - Powerset lattice: set of all subsets of $S$, ordered somehow (often by $\subseteq$)
      - $T = \emptyset$, $\perp = S$ or vice versa
      - Collecting analysis
        - Isomorphic to tuple of booleans indicating membership in subset of elements of $S$

Tuples of Lattices

- Often useful to break complex lattice into a tuple of lattices, one per variable analyzed
  - $D_T = <S_T, \leq_T> = <S, \leq>^N$
  - $S_T = S_T \times S_T \times \ldots \times S_T$
    - $\leq_T$ pointwise ordering
  - $T_T = <T_D, \leq_T>$, bottom tuple of bottoms
    - $\text{Height}(D_T) = N \times \text{height}(D)$
    - Example?
Analysis of Loops with Lattices

F = flow function for loop body
F(info-at-loop-head) = info at back edge:

F₀ = d₁⁺⁺ᵀ
F₁ = d₁⁺⁺ᵀ(F(F₀))
F₂ = d₁⁺⁺ᵀ(F(F₁))
Fₖ₊₁ = d₁⁺⁺ᵀ(F(Fₖ))

Repeat until Fₖ₊₁ = Fₖ