Intermediate Representations

CS2210
Lecture 11

Reading & Topics

- Muchnick: chapter 6
- Topics today:
  - Intermediate representations
  - Automatic code generation with pattern matching
  - Optimization Overview
  - Control Flow Analysis (start)

Intermediate Representations

- Make optimizer independent of source and target language
- Usually multiple levels
  - HIR = high level encodes source language semantics
  - Can express language-specific optimizations
  - MIR = representation for multiple source and target languages
  - Can express source/target independent optimizations
  - LIR = low level representation with many specifics to target
  - Can express target-specific optimizations
IR Goals
- Primary goals
  - Easy & effective analysis
  - Few cases
  - Support for things of interest
  - Easy transformations
  - General across source / target languages
- Secondary goals
  - Compact in memory
  - Easy to translate from / to
  - Debugging support
  - Extensible & displayable

High-Level IRs
- Abstract syntax tree + symbol table
  - most common
- LISP S-expressions

Medium-level IRs
- Represent source variables + temporaries and registers
- Reduce control flow to conditional + unconditional branches
- Explicit operations for procedure calls and block structure
- Most popular: three address code
  - t1 := t2 op t3 (address at most 3 operands)
  - if t goto L
  - t1 := t2 < t3
Important MIRs

- **SSA = static single assignment form**
  - Like 3-address code but every variable has exactly one reaching definition
  - Makes variables independent of the locations they are in
  - Makes many optimization algorithms more effective

SSA Example

\[
\begin{align*}
  &x := u \\
  &\ldots \\
  &\ldots \quad x \ldots \\
  &x := v \\
  &\ldots \\
  &\ldots \quad x \ldots
\end{align*}
\]

Other Representations

- **Triples**
  - \( i + 1 \)
  - \( i := (1) \)
  - \( i + 1 \)
  - \( p + 4 \)
  - \( * (4) \)
  - \( p := (4) \)

- **Trees**
  - Like AST but at lower level
  - Directed Acyclic Graphs (DAGs)
  - More compact than trees through node sharing
Three Address Code Example

for i := 1 to 10 do
    a[i] := b[i] + 5;
end

Representant Components

- Operations
- Dependences between operations
  - Control dependences: sequencing of operations
  - Side effects of statements occur in right order
  - Data dependences: flow of values from definitions to uses
  - Values read from variable before being overwritten
- Want to represent only relevant dependences
  - Dependences constrain operations, so the fewer the better

Representing Control Dependence

- Implicit in AST
- Explicit as Control Flow Graphs (CFGs)
  - Nodes are basic blocks
  - Edges represent branches (control flow between blocks)
- Fancier:
  - Control Dependence Graph
  - Value dependence graph (VDG)
  - Control dependence converted to data dependence
Data Dependence Kinds

- True (flow) dependence (read after write RAW)
  - Reflects real data flow, operands to operation
- Anti-dependence (WAR)
- Output dependence (WAW)
  - Reflects overwriting of memory, not real data flow
  - Can often be eliminated

Data Dependence Example

x := 3
if q != NULL then
  y := x + 2
  w := *q
  x := z * 10
else
  x := 4
endif

(1) x := 3
(2) q != 0?
(3) y := x + 2
(4) w := *q
(5) x := z * 10
(6) x := 4

Representing Data Dependences (within bb’s)

- Sequence of instructions
  - Simple
  - Easy analysis
  - But: may overconstrain operation order
- Expression tree / DAG
  - Directly captures dependences in block
  - Supports local CSE (common subexpression elimination)
  - Can be compact
  - Harder to analyze & transform
  - Eventually has to be linearized
Representing Data Dependences (across blocks)

- Implicit via def-use
  - Simple
  - Makes analysis slow (have to compute dependences each time)
- Explicit: def-use chains
  - Fast
  - Space-consuming
    - Has to be updated after transformations
- Advanced options:
  - SSA
  - VDGs
  - Dependence flow graphs (DFGs)

Manual vs. Automatic Code Generation

- Manual
  - Flexible
  - Can handcraft highly optimized code
  - Laborious to adapt to new target
- Automatic
  - Easy, fast adaptation to new targets
  - Less flexible
  - Stylized code generation can result in worse code quality

Approaches

- Graham-Glanville Code Generators
  - Pattern matching and rules similar to SLR(1) parsing rules
  - Reductions generate target code
- Attribute grammars
  - Semantic actions generate target code
- Generalized tree pattern matching
  - Match expression trees and generate corresponding code templates
Graham-Glanville Code Generation

- Uses Context-free-grammar-like rules to represent operations with corresponding instruction templates
- Semantic actions generate the code

Components
- IR transformations
  - Transforms IR to representation closer to target and appropriate for pattern matcher
- Pattern matcher
  - Uses pattern matching to find appropriate reduction
- Instruction generator
  - Generates code sequences

Machine Descriptions

<table>
<thead>
<tr>
<th>Rule</th>
<th>Template</th>
</tr>
</thead>
<tbody>
<tr>
<td>r.1</td>
<td>r.2 =&gt; r.1 or r.1,0,r.2</td>
</tr>
<tr>
<td>r.3</td>
<td>r.1 + r.2 =&gt; + r.1 r.2 add r.1,r.2,r.3</td>
</tr>
<tr>
<td>r.3</td>
<td>r.1 + k.2 =&gt; + r.1 k.2 add r.1,k.2,r.3</td>
</tr>
<tr>
<td>r.2</td>
<td>{r.2} =&gt; r.1 ldr {r.1},r.2</td>
</tr>
<tr>
<td>i</td>
<td>i =&gt; r.2 r.1 st r.1 [r.2]</td>
</tr>
</tbody>
</table>

Matching example:
<-- *r1 2 + '*r3 3 (prefix notation for: *(r1+2) := *r3+3)
<-- r2 + 'r3 3 add r1,2,r2
<-- r2 + r4 3 ldr {r3},r4
<-- r2 r5 add r4,3,r5
st r2,[r5]  

Tree Pattern Matching

- AKA BURS-style code generation
- BURS = Bottom Up Rewrite System
- Replaces (sub) trees by other trees (nodes)
  - Each pattern match rule has an associated (code generation) action and cost
- Try to find minimal cost tree cover to generate locally optimal code
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
  - Even 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
  \[ x := 5 \Rightarrow y := x \]
  \[ y := x + 2 \quad y := 5 + 2 \]
  \[ y := 7 \]

- Copy propagation
  \[ x := y \Rightarrow x := y \]
  \[ w := w + x \quad w := w + y \]

- Strength reduction
  \[ x := y \times 4 \Rightarrow x := y \ll 2 \]

- Constant propagation
  \[ x := 5 \Rightarrow x := 5 \]
  \[ y := x + 2 \quad y := 5 + 2 \]
  \[ y := 7 \]

Catalog (2)

- Common Subexpression Elimination (CSE)
  \[ x := a + b \Rightarrow x := a + b \]
  \[ y := a + b \quad y := x \]
  - Can also eliminate redundant memory references, branch tests
- Partial Redundancy Elimination (PRE)
  - Like CSE but earlier expression available only along some path

Catalog (3)

- Pointer analysis
  \[ p := &x \Rightarrow p = &x \]
  \[ p := 5 \quad *p := 5 \]
  \[ y := x + 1 \quad y := 6 \]
- Dead assignment elimination
  \[ x := y * z \]
  \[ ... /\* no use of x */ \]
  \[ x := 6 \]
- Dead code elimination
  \[ \text{if (false) then} \]
- Integer range analysis
  \[ \text{for} (i := 0; i < 10; i++) \{
    \text{if} (i > 10) \text{goto error} \\
    a[i] := 0;
  \}

CS2210 Compiler Design 2004/5
Loop Optimizations (1)

- Loop-invariant code motion
  for j := 1 to 10
  for i := 1 to 10
    a[i] := a[i] + b[j]
  for j := 1 to 10
  t := b[j]
  for i := 1 to 10
    a[i] := a[i] + t

- Induction variable elimination
  for i := 1 to 10
  a[i] := a[i] + 1
  *p := *p + 1
  a[] is several instructions *p is one

Loop Optimizations (2)

- Loop unrolling
  for i := 1 to N
  a[i] := a[i] + 1
  for i := 1 to N by 4
    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

  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 CFG

Dominators & Postdominators

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

- \( p \ p\text{dom} \ i \iff \) every possible execution path from node \( i \) to exit includes \( p \)
- Dual relation: \( i \ p\text{dom} \ p \) in CFG with edges reversed and entry and exit switched
- \( a \text{ strictly dominates } b \iff a \ p\text{dom} b \) and \( a \neq b \)

---

**Dominator Example**

![Dominator Example Diagram]

---

**Computing Dominators (easy but slow)**

- Initialize \( \text{dom}(i) = \text{set of all nodes for } i \neq \text{entry} \), \( \text{dom(entry)} = \{\text{entry}\} \)
- While changes occur do
  - \( \text{dom}(i) = \{i\} \cup (\text{dom}(i) \cap \text{dom}(	ext{pred}(i))) \) for all predecessors of \( i \)
- Works fastest if nodes are processed in DFS order
  - \( O(n^2 e) \) complexity, \( n \) number of nodes, \( e \) number of edges
Computing IDOM

- Compute dominators
- \( \text{tmp}(i) := \text{dom}(i) - \{i\} \)
- Remove all \( n \in \text{tmp}(i) \) from \( \text{tmp}(i) \) for which \( \text{dom}(n) \neq \{n\} \)

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 showing headers and preheaders with nodes B1, B2, B3.

---

**Natural Loop Properties**

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

Diagram showing natural loops with the same header.

---

**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

Diagram showing strongly connected components.
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 flowgraph.
- Leaves of control tree are basic blocks.
- Nodes between root and leaves represent regions.
- Edges represent relationships between abstract node and its descendants.
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