CS415 Compilers

Intermediate representations

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University
Announcements

- Second project - Deadline extension? Current deadline: Monday, April 19

- Sixth homework has been posted; deadline April 21

- Quiz 3 has been posted. Deadline: Friday, April 23 (60 minutes)

- Third project - Local optimization pass for ILOC; will be posted next Monday, April 19. Due on Monday, May 3

- Quiz 4 and seventh homework will be posted during last week of classes (no eighth homework!)

- "Finals" Quiz during finals week
Intermediate Representations
(EaC Chapter 5)

- Front end - produces an intermediate representation ($IR$)
- Middle end - transforms the $IR$ into an equivalent $IR$ that runs more efficiently
- Back end - transforms the $IR$ into native code

- $IR$ encodes the compiler’s knowledge of the program
- Middle end usually consists of several passes
Intermediate Representations

• Decisions in IR design affect the speed and efficiency of the compiler

• Some important IR properties
  → Ease of generation
  → Ease of manipulation
  → Size
  → Level of abstraction

• The importance of different properties varies between compilers
  → Selecting an appropriate IR for a compiler is critical
Three major categories

- **Structural**
  - Graphically oriented
  - Heavily used in source-to-source translators
  - Tend to be large
  - Examples: Trees, DAGs

- **Linear**
  - Pseudo-code for an abstract machine
  - Level of abstraction varies
  - Simple, compact data structures
  - Easier to rearrange
  - Examples: 3 address code, Stack machine code

- **Hybrid**
  - Combination of graphs and linear code
  - Example: Control-flow graph
Level of Abstraction

- The level of detail exposed in an IR influences the profitability and feasibility of different optimizations.
- Two different representations of an array reference:

<table>
<thead>
<tr>
<th>High level AST: Good for memory disambiguation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low level linear code: Good for address calculation</td>
</tr>
</tbody>
</table>

```plaintext
subscript

A    i    j

loadI 1  => r₁
sub  r_j, r₁ => r₂
loadI 10 => r₃
mult r₂, r₃ => r₄
sub  r_i, r₁ => r₅
add  r₄, r₅ => r₆
loadI @A => r₇
Add  r₇, r₆ => r₈
load  r₈ => r_{Aij}
```
Level of Abstraction

- Structural IRs are usually considered high-level.
- Linear IRs are usually considered low-level.
- Not necessarily true:

```
load @A
loadArray A,i,j
```

Low level AST

```
+  
| +  
| | +  
| | @A  
| *  
|-- -  
|   |  
|   |  
| 10 -  
|iji
```

High level linear code
Abstract Syntax Tree

An abstract syntax tree is the procedure's parse tree with the nodes for most non-terminal nodes removed.

- Can use linearized form of the tree
  → Easier to manipulate than pointers
  \[ x \ 2 \ y \ * \ - \] in postfix form
  \[ - * \ 2 \ y \ x \] in prefix form
- S-expressions are (essentially) ASTs (remember functional languages such as Scheme or Lisp!)
A directed acyclic graph (DAG) is an AST with a unique node for each value.

- Makes sharing explicit
- Encodes redundancy

Same expression twice means that the compiler might arrange to evaluate it just once!
Stack Machine Code

Originally used for stack-based computers, now Java

• Example:
  \[ x - 2 \times y \]

  becomes

  push x
  push 2
  push y
  multiply
  subtract

Advantages

• Compact form
• Introduced names are *implicit*, not *explicit*
• Simple to generate and execute code

Useful where code is transmitted over slow communication links (*the net*)

Implicit names take up no space, where explicit ones do!
Several different representations of three address code

- In general, three address code has statements of the form:
  \[ x \leftarrow y \; op \; z \]
  
  With 1 operator (\( op \)) and, at most, 3 names (\( x, y, z \))

Example:

\[
\begin{align*}
  z & \leftarrow x \\
  \text{2 * y becomes} & \\
  t & \leftarrow 2 \times y \\
  z & \leftarrow x - t
\end{align*}
\]

Advantages:

- Resembles many machines
- Introduces a new set of names
- Compact form
Naïve representation of three address code

- Table of $k \times 4$ small integers
- Simple record structure
- Easy to reorder
- Explicit names

RISC assembly code (not ILOC)

Subtract $r_5$, $r_4$, $r_3$

<table>
<thead>
<tr>
<th>load</th>
<th>r1, y</th>
</tr>
</thead>
<tbody>
<tr>
<td>loadI</td>
<td>r2, 2</td>
</tr>
<tr>
<td>mult</td>
<td>r3, r2, r1</td>
</tr>
<tr>
<td>load</td>
<td>r4, x</td>
</tr>
<tr>
<td>sub</td>
<td>r5, r4, r3</td>
</tr>
</tbody>
</table>

Quadruples

<table>
<thead>
<tr>
<th>load</th>
<th>1</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>loadI</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>mult</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>load</td>
<td>4</td>
<td>x</td>
</tr>
<tr>
<td>sub</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>
Three Address Code: Triples

- Index used as implicit name
- 25% less space consumed than quads
- Much harder to reorder

<table>
<thead>
<tr>
<th></th>
<th>(1)</th>
<th>(2)</th>
<th>(3)</th>
<th>(4)</th>
<th>(5)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>load</td>
<td>y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>loadI</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>mult</td>
<td>(1)</td>
<td>(2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>load</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sub</td>
<td>(4)</td>
<td>(3)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Implicit names take no space!
Control-flow Graph (CFG)

Models the transfer of control in the procedure
- Nodes in the graph are basic blocks
  - Can be represented with quads or any other linear representation
- Edges in the graph represent control flow

Example

```
if (x = y)
  a ← 2
  b ← 5
  c ← a + b
  a ← 3
  b ← 4
```

Basic blocks — Maximal length sequences of straight-line code
Static Single Assignment Form (SSA)

- The main idea: each name defined exactly once in program
- Introduce $\phi$-functions to make it work

**Original**

- $x \leftarrow \ldots$
- $y \leftarrow \ldots$
- while ($x < k$)
  - $x \leftarrow x + 1$
  - $y \leftarrow y + x$

**SSA-form**

- $x_0 \leftarrow \ldots$
- $y_0 \leftarrow \ldots$
- if ($x_0 > k$) goto next loop:
  - $x_1 \leftarrow \phi(x_0, x_2)$
  - $y_1 \leftarrow \phi(y_0, y_2)$
  - $x_2 \leftarrow x_1 + 1$
  - $y_2 \leftarrow y_1 + x_2$
  - if ($x_2 < k$) goto loop
  - next: ...$

**Strengths of SSA-form**

- Sharper analysis
- “minimal” $\phi$-functions placement is non-trivial
- (sometimes) faster algorithms
Things to do and next class

Procedure abstractions

Runtime environments

Work on the project!