Intermediate Representations

Copyright 2003, Keith D. Cooper, Ken Kennedy & Linda Torczon, all rights reserved.
Students enrolled in Comp 412 at Rice University have explicit permission to make copies of these materials for their personal use.
Intermediate Representations

- 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
  → Procedure size
  → Freedom of expression
  → Level of abstraction

• The importance of different properties varies between compilers
  → Selecting an appropriate IR for a compiler is critical
Types of Intermediate Representations

Three major categories

• Structural
  → Graphically oriented
  → Heavily used in source-to-source translators
  → Tend to be large

• Linear
  → Pseudo-code for an abstract machine
  → Level of abstraction varies
  → Simple, compact data structures
  → Easier to rearrange

• Hybrid
  → Combination of graphs and linear code
  → Example: control-flow graph

Examples:
  Trees, DAGs
  3 address code
  Stack machine code
  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:

  High level AST:
  Good for memory disambiguation

  Low level linear code:
  Good for address calculation

```plaintext
loadI 1   => r_1
sub  r_j, r_1 => r_2
loadI 10  => r_3
mult r_2, r_3 => r_4
sub  r_i, r_1 => r_5
add  r_4, r_5 => r_6
loadI @A  => r_7
Add  r_7, r_6 => r_8
load  r_8   => r_{Aij}
```
Level of Abstraction

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

Low level AST:
```
load
```

High level linear code:
```
loadArray A, i, j
```
Abstract Syntax Tree

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

\[
\begin{align*}
&x - 2 \times y \\
\end{align*}
\]

- 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
Directed Acyclic Graph

A directed acyclic graph (DAG) is an AST with a unique node for each value

- Makes sharing explicit
- Encodes redundancy

$$z \leftarrow x - 2 \ast y$$
$$w \leftarrow x / 2$$

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!
Three Address Code

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 \((\textit{op})\) and, at most, 3 names \((x, y, \& z)\)

Example:

\[ z \leftarrow x - 2 * y \]

becomes

\[ t \leftarrow 2 * y \]

\[ z \leftarrow x - t \]

Advantages:

• Resembles many machines
• Introduces a new set of names
• Compact form
Three Address Code: Quadruples

Naïve representation of three address code

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

```
load r1, y
loadI r2, 2
mult r3, r2, r1
load r4, x
sub r5, r4, r3
```

RISC assembly code

<table>
<thead>
<tr>
<th>Quadruples</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>loadi</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mult</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>load</td>
<td>4</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The original FORTRAN compiler used “quads”
Three Address Code: Triples

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

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

Implicit names take no space!
Three Address Code: Indirect Triples

- List first triple in each statement
- Implicit name space
- Uses more space than triples, but easier to reorder

<table>
<thead>
<tr>
<th>(100)</th>
<th>(100)</th>
<th>load</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>(105)</td>
<td>(101)</td>
<td>loadI</td>
<td>2</td>
</tr>
<tr>
<td>(102)</td>
<td>mult</td>
<td>(100)</td>
<td>(101)</td>
</tr>
<tr>
<td>(103)</td>
<td>load</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>(104)</td>
<td>sub</td>
<td>(103)</td>
<td>(102)</td>
</tr>
</tbody>
</table>

- Major tradeoff between quads and triples is compactness versus ease of manipulation
  - In the past compile-time space was critical
  - Today, speed may be more important
The main idea: each name defined exactly once

Introduce $\phi$-functions to make it work

**Static Single Assignment Form**

<table>
<thead>
<tr>
<th>Original</th>
<th>SSA-form</th>
</tr>
</thead>
</table>
| $x \leftarrow 0$
$y \leftarrow 0$
while $(x < k)$
  $x \leftarrow x + 1$
  $y \leftarrow y + x$ | $x_0 \leftarrow 0$
$y_0 \leftarrow 0$
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: $\ldots$ |

**Strengths of SSA-form**

- Sharper analysis
- $\phi$-functions give hints about placement
- (sometimes) faster algorithms
Two Address Code

• Allows statements of the form
  \[ x \leftarrow x \ op \ y \]
  Has 1 operator (op) and, at most, 2 names (x and y)

Example:
  \[ z \leftarrow x - 2 \ast y \]
  becomes

  \[
  \begin{align*}
  t_1 & \leftarrow 2 \\
  t_2 & \leftarrow \text{load } y \\
  t_2 & \leftarrow t_2 \ast t_1 \\
  z & \leftarrow \text{load } x \\
  z & \leftarrow z - t_2 
  \end{align*}
  \]

• Can be very compact

Problems
• Machines no longer rely on destructive operations
• Difficult name space
  → Destructive operations make reuse hard
  → Good model for machines with destructive ops (PDP-11)
Control-flow Graph

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
  a ← 3
  b ← 4
  c ← a * b
```
Using Multiple Representations

- Repeatedly lower the level of the intermediate representation
  - Each intermediate representation is suited towards certain optimizations
- Example: the Open64 compiler
  - WHIRL intermediate format
    - Consists of 5 different IRs that are progressively more detailed
Memory Models

Two major models

• Register-to-register model
  → Keep all values that can legally be stored in a register in registers
  → Ignore machine limitations on number of registers
  → Compiler back-end must insert loads and stores

• Memory-to-memory model
  → Keep all values in memory
  → Only promote values to registers directly before they are used
  → Compiler back-end can remove loads and stores

• Compilers for RISC machines usually use register-to-register
  → Reflects programming model
  → Easier to determine when registers are used
The Rest of the Story...

Representing the code is only part of an IR

There are other necessary components

- Symbol table (already discussed)
- Constant table
  - Representation, type
  - Storage class, offset
- Storage map
  - Overall storage layout
  - Overlap information
  - Virtual register assignments