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
  - 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
  
  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:
  
  ```
  loadI 1 => r_1
  sub r_3, r_1 => r_2
  loadI 10 => r_3
  mul r_2, r_3 => r_4
  sub r_1, 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}
  ```

High level AST: Good for memory disambiguation

Low level linear code: Good for address calculation

Level of Abstraction

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

Low level AST  
```
load
+ @A
* 10 
- 
+ 
* 
1
```

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

```
-  
x  *
  2  y
```

\[ x - 2 \times y \]

- 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

\[ z \leftarrow x - 2 \times y \]
\[ w \leftarrow x / 2 \]

- 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 \textit{implicit}, not \textit{explicit}
- Simple to generate and execute code

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

Three Address Code

Several different representations of three address code

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

Example:

\[ z \leftarrow x - 2 \times y \]
becomes

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

load 1 Y
load 2 2
mult 3 2 1
load 4 X
sub 5 4 2

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

(1) load y
(2) loadI 2
(3) mult (1) (2)
(4) load x
(5) sub (4) (3)

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

• List triples in a statement list data structure
• Implicit name space
• Uses more space than triples, but easier to reorder

<table>
<thead>
<tr>
<th>stmt array</th>
</tr>
</thead>
<tbody>
<tr>
<td>(103)</td>
</tr>
<tr>
<td>(101)</td>
</tr>
<tr>
<td>(100)</td>
</tr>
<tr>
<td>(102)</td>
</tr>
<tr>
<td>(104)</td>
</tr>
</tbody>
</table>

• Major tradeoff between quads and triples is compactness versus ease of manipulation (note: multiple occurrences of same statement in stmt array is possible)
  → compile-time space critical?
  → compilation speed more important?

Static Single Assignment Form (SSA)

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

Original          SSA-form

\[
\begin{align*}
x & \leftarrow \ldots \\
y & \leftarrow \ldots \\
\text{while} \ (x < k) & \\
x & \leftarrow x + 1 \\
y & \leftarrow y + x
\end{align*}
\]

\[
\begin{align*}
x_0 & \leftarrow \ldots \\
y_0 & \leftarrow \ldots \\
\text{if} \ (x_0 > k) & \text{goto next} \\
\text{loop:} & \\
x_1 & \leftarrow \phi (x_0, x_2) \\
y_1 & \leftarrow \phi (y_0, y_2) \\
x_2 & \leftarrow x_1 + 1 \\
y_2 & \leftarrow y_1 + x_2 \\
\text{if} \ (x_2 < k) & \text{goto loop} \\
\text{next:} & \\
& \ldots
\end{align*}
\]

Strengths of SSA-form
• Sharper analysis
• \( \phi \)-functions give hints about placement
• (sometimes) faster algorithms
Two Address Code

- Allows statements of the form
  \[ x ← x \text{ op } y \]
  Has 1 operator (op) and, at most, 2 names (x and y)

Example:
\[ z ← x - 2 * y \] becomes
\[ t_1 ← 2 \]
\[ t_2 ← \text{load } y \]
\[ t_2 ← t_2 * t_1 \]
\[ z ← \text{load } x \]
\[ z ← z - t_2 \]

- 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 (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
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

Virtual Machines

• Can interpret IR using virtual machine
• Examples
  → P-code for Pascal
  → postscript for display devices
  → Java byte code for everywhere
• Result
  → easy & portable
  → much slower
• Just-in-time compilation (JIT)
  → begin interpreting IR
  → find performance critical section(s)
  → compile section(s) to native code
  → ...or just compile entire program
  → compilation time becomes execution time
Java Virtual Machine (JVM)

- The JVM consists of four parts
- Memory
  - Stack (for function call frames)
  - Heap (for dynamically allocated memory)
  - Constant pool (shared constant data)
  - Code segment (instructions of class files)
- Registers
  - Stack pointer (SP), local stack pointer (LSP), program counter (PC)
- Condition codes
  - Stores result of last conditional instruction
- Execution unit
  1. Reads current JVM instruction
  2. Change state of virtual machine
  3. Increment PC (modify if call, branch)

Java Byte Codes

### Arithmetic instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ineg</td>
<td>[...i]</td>
<td>-&gt; [...-i]</td>
</tr>
<tr>
<td>iadd</td>
<td>[...i1,i2]</td>
<td>-&gt; [...i1+i2]</td>
</tr>
<tr>
<td>isub</td>
<td>[...i1,i2]</td>
<td>-&gt; [...i1-i2]</td>
</tr>
<tr>
<td>imul</td>
<td>[...i1,i2]</td>
<td>-&gt; [...i1*i2]</td>
</tr>
<tr>
<td>idiv</td>
<td>[...i1,i2]</td>
<td>-&gt; [...i1/i2]</td>
</tr>
</tbody>
</table>

### Direct instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iinc k a</td>
<td>[...]</td>
<td>-&gt; [...]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>local[k] &lt;- local[k]+a</td>
</tr>
</tbody>
</table>
Java Byte Codes

Branch instructions

goto L \ [\ldots \to \ldots] \quad \text{branch to } L
ifeq L \ [\ldots:i \to \ldots] \quad \text{branch if } i = 0
ifne L \ [\ldots:i \to \ldots] \quad \text{branch if } i \neq 0
ifnull L \ [\ldots:o \to \ldots] \quad \text{branch if } o \text{ is null}
ifnonnull L \ [\ldots:o \to \ldots] \quad \text{branch if } o \neq \text{null}
if_icmpeq L \ [\ldots:i1:i2 \to \ldots] \quad \text{branch if } i1 = i2
if_icmpne L \ [\ldots:i1:i2 \to \ldots] \quad \text{branch if } i1 \neq i2
if_icmp gt L \ [\ldots:i1:i2 \to \ldots] \quad \text{branch if } i1 > i2
if_icmplt L \ [\ldots:i1:i2 \to \ldots] \quad \text{branch if } i1 < i2
if_acmpeq L \ [\ldots:o1:o2 \to \ldots] \quad \text{branch if } o1 = o2
if_acmpne L \ [\ldots:o1:o2 \to \ldots] \quad \text{branch if } o1 \neq o2

Java Byte Codes

Constant loading

iconst_0 \ [\ldots \to \ldots:0]
icont_1 \ [\ldots \to \ldots:1]
aconst_null \ [\ldots \to \ldots:\text{null}]
ldc_int i \ [\ldots \to \ldots:i]
ldc_string s \ [\ldots \to \ldots:\text{str}(s)]

Locals operations

iload k \ [\ldots \to \ldots:local[k]]
aload k \ [\ldots \to \ldots:local[k]]
istore k \ [\ldots:i \to \ldots:local[k] := i]
astore k \ [\ldots:o \to \ldots:local[k] := o]
Java Byte Codes

Stack operations
- dup: \([...;v] \rightarrow [...;v,v]\)
- pop: \([...;v] \rightarrow [...]\)
- swap: \([...;v1,v2] \rightarrow [...;v2,v1]\)

Functions
- invoke: \([...;args] \rightarrow [...]\) push stack frame, ...
- ireturn: \([...;i] \rightarrow [...]\) ret i, pop stack frame
- areturn: \([...;o] \rightarrow [...]\) ret o, pop stack frame
- return: \([...] \rightarrow [...]\) pop stack frame

Java Byte Code Interpreter

```java
pc = code.start
while (true) {
    new_pc = pc + inst.len(code[pc]);
    switch (opcode(code[pc])) {
        case iconst_1:
            push(1); break;
        case iload:
            push(local[code[pc+1]]); break;
        case istore:
            t ← pop();
            local[code[pc+1]] ← t; break;
        case iadd:
            t1 ← pop(); t2 ← pop();
            push(t1 + t2); break;
        case ifeq:
            t ← pop();
            if (t = 0) new_pc = code[pc+1]; break;
            ...
    } new_pc = code[pc+1];
    pc ← new_pc;
}
```