Intermediate Representations
*The Glue That Holds a Compiler Together*

Comp 412

Copyright 2017, Keith D. Cooper & 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.
Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved.

*Chapter 5 in EaC2e*
Where In The Course Are We?

**Obvious answer:** *at the start of Chapter 5 in EaC2e*

**More important answer**

- We are on the cusp of the art, science, & engineering of compilation
- Scanning & parsing are applications of automata theory
- Context-sensitive analysis (**AHSDT**) is mostly software engineering
- The mid-section of the course will focus on issues where the compiler writer needs to choose among alternatives
  - The choices matter; they affect the quality of compiled code
  - There may be no “best answer” or “best practice”

*To my mind, the fun begins at this point*
Intermediate Representations

Front End emits IR for the rest of the compiler to use

- Scanner & parser work from the syntax of the source code
  - Parser emits IR using AHSDT or some similar technique
- Rest of the compiler works from IR
  - Analyzes IR to learn about the code
  - Transforms IR to improve final code
- IR determines what a compiler can do to the code
  - Can only manipulate details that are represented in the IR
Intermediate Representations

**IR is the vehicle that carries information between phases**

- **Front end**: produces an IR version of the code
- **Optimizer**: transforms the IR into an equivalent IR that runs faster
  - Each “pass” reads and writes IR
- **Back end**: systematically transforms the IR into native code

**IR determines both the compiler’s ambition & its chances for success**

- The compiler’s knowledge of the code is encoded in the IR
- The compiler can only manipulate what is represented by the IR
Intermediate Representations

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

Some important IR properties

• Ease of generation
• Ease of manipulation
• Cost of manipulation
• Procedure size
• Expressiveness
• Level of abstraction

Example

Ease of Manipulation: in your register allocator lab, some of you used Java’s ArrayList class. ArrayList provides many useful features that you did not need to implement yourself.

Cost of Manipulation: ArrayList is efficient, unless you are inserting items into the middle of a long list, as occurs when inserting spills and restores.

Impact on the Compiler: to combat the cost of insertion, some of you printed the output on the fly rather than updating the IR

The importance of different properties varies between compilers

⇒ Selecting an appropriate IR for a compiler is critical
Intermediate Representations

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

Some important IR properties

• Ease of generation
• Ease of manipulation
• Cost of manipulation
• Procedure size
• Expressiveness
• Level of abstraction

Example

Expressiveness: a compiler with a near-source level IR may have trouble representing the results of some optimizations.

\[
\begin{align*}
\text{for } i &= 0 \text{ to } n \\
\text{for } j &= 0 \text{ to } m \\
a[i,j] &= 0 \\
p &= \&a[0,0] \\
\text{for } i &= 0 \text{ to } n \times m \\
*p++ &= 0
\end{align*}
\]

Simple initialization After OSR

The implementation of operator strength reduction (OSR) can only produce this result if the IR can represent \(\ast p++\). (§ 10.7.2 in EaC2e.)

The importance of different properties varies between compilers

⇒ Selecting an appropriate IR for a compiler is critical
Intermediate Representations

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

Some important IR properties

- Ease of generation
- Ease of manipulation
- Cost of manipulation
- Procedure size
- Expressiveness
- Level of abstraction

Example

*Level of Abstraction:* copying a string is a complex operation that involves an internal loop. Explicitly representing the loop and its details exposes all those details to uniform optimization (a good thing).

Explicitly representing the loop makes it difficult to move the copy to another location in the code — moving control flow constructs is difficult, at best.

Representing the copy as a single operation, like the S370 `mvcl` makes it easy to move.

The importance of different properties varies between compilers

⇒ Selecting an appropriate IR for a compiler is critical
Intermediate Representations

Today, we will focus on representing the operations in the code
• Arithmetic expressions
• Assignment
• Control-flow in the program

Later, we will come back and talk about representing names & objects
• Symbol tables
• Renaming
• Storage models
Taxonomy of Intermediate Representations

Three major categories

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

• Linear IRs
  – 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 IRs
  – Combination of graphs and linear code
  – Example: control-flow graph
  Examples: Control-flow graph
           SSA Form
Level of Abstraction

The level of detail exposed in an IR influences the profitability and feasibility of different optimizations.

Here are two different representations of an array reference:

High level AST
Good for memory disambiguation

Low level linear code
Good for optimizing the address calculation

loadI 1 => r₁
sub r₁, r₁ => r₂
loadI 10 => r₃
mult r₂, r₃ => r₄
sub r₁, r₁ => r₅
add r₄, r₅ => r₆
loadI @A => r₇
add r₇, r₆ => r₈
load r₈ => rₐᵢⱼ
Level of Abstraction

People tend to confuse level of abstraction with structure

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

Low level AST

![Low level AST diagram]

High-level linear code

loadArray A, i, j

In Chapter 11 of EaC2e, we will see trees that have a lower level of abstraction than the machine code
A syntax tree represents the front ends’ parse of the code, in detail

Syntax trees are often used in source-to-source systems

- Captures the precise (syntactic) form of the input program
- Has all of the detail that you could need
  - Compiler generated all the detail it has from the parse

Syntax trees tend to be inefficient

- Lots of unnecessary nodes and edges
- Lots of implicit detail that might be useful to represent explicitly

Parse tree for \( x - 2 \ast y \)

Syntax trees can be represented with a linear notation (e.g., prefix or postfix)
Abstract Syntax Tree

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

- **ASTs** are space efficient trees that capture most of the interesting information found in a syntax tree.
  - Can regenerate source code in a treewalk, with a little cleverness.
  - Many fewer nodes and edges than in a syntax tree.

- S-expressions in Scheme or Lisp, are (essentially) **ASTs**.

In practice, **ASTs** tend to be large — not because they must, but they grow (see the digression on page 228 in EaC2e).
A directed acyclic graph (DAG) is an AST with a unique node for each value (an AST with sharing)

- Makes sharing explicit
- Encodes redundancy

If the compiler uses graphical IRs, a **DAG** is a natural way to represent redundancy.

With two copies of the same expression, the compiler may be able to arrange the code to evaluate it only once.
Implementing Trees

In earlier courses, you learned to build purpose-built trees (or to use some generic library)

- Nodes connected by edges
- Nodes of various types and arities
- Allocating nodes of different sizes complicates both allocation & fragmentation in the heap (malloc())

Knuth showed that you can map any arbitrary tree onto a binary tree

- Allocate uniform-size, binary nodes
- Two “pointers”: child and sibling
- Simplify allocation and traversal

1 See Knuth Volume 1, pp. 332-334; also EaC2e pp. 744-746
Compilers use a dependence graph to understand & preserve the flow of values in a block.

Dependence Graph

1. loadI 8 => r1
2. loadI 12 => r2
3. mult r1, r2 => r3
4. add r1, r3 => r4

Original Code

Dependence Graph

- Typically built as a secondary IR for scheduling or optimization
- Exposes the constraints on execution order due to flow of value
- Critical for detection of parallelism, for reordering iteration spaces of loops (memory hierarchy optimization), & for scheduling

In Lab 3, you will build dependence graphs.
Digression (or Rant)

The role of string data

Principle:

*For the sake of compactness and efficiency, the compiler should almost never store a string value in the IR*

- Strings are expensive to compare
- Strings are large relative to their information content
- In most cases, we need just one copy of a string

How to handle strings in your IR

- Use a hash table or map to convert each string to a small integer
- Represent the string with the integer
- Compare strings as integers
Three major categories

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

Examples:
Trees, DAGs

• Linear IRs
  – 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 IRs
  – Combination of graphs and linear code
  – Example: control-flow graph

Examples:
Control-flow graph
SSA Form
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 \( \text{(op)} \) and, at most, 3 names \( (x, y, \& z) \)

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

Advantages:
• Resembles many real machines
• Introduces a new set of names
• Compact form

See Lab 1 and Lab 3

The concept of “three address code” has many implementations.
Three Address Code: As Quadruples

Naïve representation of three address code

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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Op_1</th>
<th>Op_2</th>
<th>Op_3</th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
<td>1</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>loadI</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>mult</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>load</td>
<td>4</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>sub</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

RISC assembly code

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

Quadruples

The original FORTRAN compiler used “quads”

Store opcode & operands as small integers, of course.
Three Address Code: As Triples

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

<table>
<thead>
<tr>
<th>Implicit Name</th>
<th>Opcode</th>
<th>Op₁</th>
<th>Op₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>load</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>(2)</td>
<td>loadI</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>(3)</td>
<td>mult</td>
<td>(1)</td>
<td>(2)</td>
</tr>
<tr>
<td>(4)</td>
<td>load</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>(5)</td>
<td>sub</td>
<td>(4)</td>
<td>(3)</td>
</tr>
</tbody>
</table>

Remember, for a long time, 640 KB was a lot of RAM
Three Address Code: As Indirect Triples

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

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

Standard trick: when you need the ability to rearrange objects in memory, add a level of indirection.
Two-Address Code

Two-address code allows statements of the form

\[ x \leftarrow x \; op \; y \]

Each operation has 1 operator \( op \) and, at most, 2 names \( x \) and \( y \)

**Example**

\[ z \leftarrow x - 2 \times y \quad \text{becomes} \quad t_1 \leftarrow 2 \]
\[ t_2 \leftarrow \text{load } y \]
\[ t_2 \leftarrow t_2 \times t_1 \]
\[ z \leftarrow \text{load } x \]
\[ z \leftarrow z - t_2 \]

**Problems**

- Difficult name space
  - Destructive operations make reuse hard
  - Good model for machines with destructive ops (PDP-11, x86)
- We would like destructive operations to become a thing of the past

We write:

\[ r_1 + r_2 \Rightarrow r_2 \]

as:

\[ \text{add } r_1, r_2 \]

Not many arguments in favor of two-address code
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 or where memory is limited

- Java bytecode was designed for transmission over slow links
- Follows a long line of bytecode-like IRs designed to be compact

In a stack machine, most operations are destructive.
Taxonomy of Intermediate Representations

Three major categories

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

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

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

Examples:
- Trees, DAGs
- 3 address code
- Stack machine code
- Control-flow graph
- SSA Form
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

\[
\begin{align*}
\text{if } (x = y) \\
\text{a } &\leftarrow 2 \\
\text{b } &\leftarrow 5 \\
\text{a } &\leftarrow 3 \\
\text{b } &\leftarrow 4 \\
\text{c } &\leftarrow \text{a } \ast \text{b}
\end{align*}
\]

Hybrid IRs

Basic blocks: Maximal length sequences of straightline code

Edges represent the branches & jumps at the ends of blocks.

Implementations:
See Figures B.3 and B.4 in Appendix B of EaC2e
Static Single Assignment Form

The Main Idea: each name defined by exactly one operation

• Introduce $\phi$-functions to make it work

<table>
<thead>
<tr>
<th>Original</th>
<th>SSA-form</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x \leftarrow \ldots$</td>
<td>$x_0 \leftarrow \ldots$</td>
</tr>
<tr>
<td>$y \leftarrow \ldots$</td>
<td>$Y_0 \leftarrow \ldots$</td>
</tr>
<tr>
<td>while ($x &lt; k$)</td>
<td>if ($x_0 &gt;= k$) goto next</td>
</tr>
<tr>
<td>$x \leftarrow x + 1$</td>
<td>loop: $x_1 \leftarrow \phi(x_0, x_2)$</td>
</tr>
<tr>
<td>$y \leftarrow y + x$</td>
<td>$y_1 \leftarrow \phi(y_0, y_2)$</td>
</tr>
<tr>
<td></td>
<td>$x_2 \leftarrow x_1 + 1$</td>
</tr>
<tr>
<td></td>
<td>$y_2 \leftarrow y_1 + x_2$</td>
</tr>
<tr>
<td></td>
<td>if ($x_2 &lt; k$) goto loop</td>
</tr>
<tr>
<td></td>
<td>next: ...</td>
</tr>
</tbody>
</table>

Strengths of SSA-form

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

SSA is often interpreted as a graph, with edges running from def to use.
Combination of IRs

Some compilers use a CFG to represent control flow and a linear IR to represent code in blocks

- This hybrid IR with the advantages of a graph
  - Easy navigation between blocks
- And the advantages of a linear IR
  - Explicit, low-level detail & operation sequence

Strengths

- Good for understanding control-flow issues
- Good for understanding flow of data (program analysis)

In lab 3, you will build and use a dependence graph, linked to a basic block. The dependence graph records the flow of values, while the block records both the details of those operations & their relative order.
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 and less abstract
    - Each successive IR focuses on a different set of challenges & opportunities
  - Translation is a monotonic lowering of level of abstraction
    - Compilers are good at lowering level of abstraction & not so good at raising it

COMP 412, Fall 2017 28
Memory Models

An IR usually incorporates a memory model, explicit or implicit.

- **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 a register-to-register model**
- Closely reflects RISC instruction sets
- Register use is explicit and well-modelled
The Rest of the Story...

Representing the code is only part of an IR

Other components are necessary

- Symbol table
  - Every name in the program
- Constant table
  - Representation, type
  - Storage class, offset
- Storage map
  - Overall storage layout
  - Overlap information
  - Virtual register assignments

We will return to these tables, their motivation, and their use over the next several lectures.