Intermediate Representations

Copyright 2019, Keith D. Cooper, Linda Torczon & Zoran Budimlić, all rights reserved.
Students enrolled in Comp 506 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.
Intermediate Representations

AHSDT (Ad-hoc syntax-directed translation) refers to the "actions" invoked by reductions in an LR(1) parser

Parser & Semantic Elaborator emit IR for the rest of the compiler to use

- Front end works from the syntax of the source code
  - Parser & Semantic Elaborator emit IR using AHSDT or 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 intermediate representation (IR)
- **Optimizer:** transforms the IR into an equivalent IR that runs faster
  - Each “pass” reads and writes IR
- **Back end:** 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 facts that are 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

The importance of different properties varies between compilers
⇒ Selecting an appropriate IR and implementing it well is crucial
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 Generation: To optimize code, the compiler must understand the flow of control: loops, if-then-else, & case statements, which are typically represented in a control-flow graph (CFG). A CFG has a node for each basic block\(^1\) and edges to represent flow of control.

To generate a CFG, however, the compiler needs to understand where blocks begin & end, and see both the source and sink of each edge. Generating a CFG on the fly is hard; most compilers need a pass over the code, in some form of IR, before they build a CFG.

The importance of different properties varies between compilers

⇒ Selecting an appropriate IR and implementing it well is crucial

\(^1\) A basic block is a maximal-length sequence of straight-line code.
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 a Java code that has a list of instructions, you might choose to use Java’s ArrayList class. ArrayList provides many useful features.

Cost of Manipulation: ArrayList is efficient, unless you are inserting items into the middle of a long list, as occurs in many optimizations.

Impact on the Compiler: if the compiler needs to reorder code or insert code, the costs can go quadratic due to the implementation of the underlying ArrayList.

The importance of different properties varies between compilers
⇒ Selecting an appropriate IR and implementing it well is crucial
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 optimization.

```plaintext
for i = 0 to n  p = &a[0,0]
    for j = 0 to m  for i = 0 to n * m
        a[i,j] = 0     *p++ = 0
```

*Simplified initialization*  
After OSR

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

**The importance of different properties varies between compilers**

⇒ Selecting an appropriate IR and implementing it well is crucial
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 and implementing it well is crucial
Taxonomy of Intermediate Representations

More on the level of abstraction

• Is the IR closer in abstraction to the source or the machine?
  ♦ “High level” ⇒ closer to source
  ♦ “Low level” ⇒ closer to machine

• The level of exposed detail influences the feasibility and profitability of different code optimizations

Example:  $A[i,j]$
Level of Abstraction

Do not confuse “graph versus code” with “high level versus low level”

- Last slide, the tree was high level and the code was low level

This situation could easily be reversed ...

In instruction selection, compilers often use trees that have a lower level of abstraction than machine code.
Taxonomy of Intermediate Representations

Three major categories of IR

• **Structural IRs**
  ♦ Graphs and trees
  ♦ Widely used in source-to-source translators
  ♦ Tend to use large amounts of memory

• **Linear IRs**
  ♦ Pseudo-code for an abstract machine
  ♦ Simple compact data structures
  ♦ Can be easy to reorder and rearrange

• **Hybrid IRs**
  ♦ Combinations of graphs and linear code
  ♦ Provide some of the advantages of both structural & linear IRs

Examples:
Trees, directed acyclic graphs, interference graph (register allocation)

Examples:
Three address code (ILOC), stack machine code (Java bytecode), register-transfer language (RTL)

Examples:
Control-flow graph, SSA form
Taxonomy of Intermediate Representations

Three major categories of IR

• **Structural IRs**
  ♦ Graphs and trees
  ♦ Widely used in source-to-source translators
  ♦ Tend to use large amounts of memory

  *Examples:* Trees, directed acyclic graphs, interference graph (register allocation)

• **Linear IRs**
  ♦ Pseudo-code for an abstract machine
  ♦ Simple compact data structures
  ♦ Can be easy to reorder and rearrange

  *Examples:* Three address code (ILOC), stack machine code (Java bytecode), register-transfer language (RTL)

• **Hybrid IRs**
  ♦ Combinations of graphs and linear code
  ♦ Provide some of the advantages of both structural & linear IRs

  *Examples:* Control-flow graph, SSA form
A parse tree, or syntax tree, represents the input program’s derivation

- Interior node for each nonterminal symbol in the derivation
- Leaf node for each word (terminal symbol in the derivation)
- Simple inorder treewalk will reproduce the original syntax

A parse tree captures every detail of the input program, but expands none of them.
- What you see is what you get
- What you see is all you have

Parse tree for $x - 2 * y$
Abstract Syntax Tree

An abstract syntax tree (AST) is the parse tree with the nodes for most nonterminal symbols removed

- Many fewer nodes (about 50%)
- Simpler structure
- Little information is lost, compared to the parse tree\(^1\)
- Can represent AST with a graph \((i.e., \text{nodes and edges})\)
- Can represent AST in a linear form
  - Easier to manipulate than pointers
    - \(2 \ y \ * \ x\) - in postfix form
    - \(- x \ * \ 2 \ y\) in prefix form
- S-expressions (scheme, lisp) are (essentially) ASTs

Some compilers build ASTs as an initial IR.

In practice, ASTs tend to be large — not because they must be, but because they grow through the addition of more and more fields on each node and edge.

See the digression on page 228 in EaC2e

\(^1\) The lost information concerns the derivation, not the meaning. Knowledge about the meaning is encoded in the structure of the AST.
Directed Acyclic Graph

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

- **DAG** makes sharing explicit
- **DAG** directly encodes redundancy
- **DAG** saves a little space
- **DAG** has no obvious linear form

Building a DAG

- Requires analysis to recognize redundancy
- Can consider **hash consing**, a scheme that folds a hash-test into node creation and returns the earlier result, if one exists
- Can consider an algorithm similar to value numbering

If the compiler **knows** that it has two copies of the same expression, it may be able to arrange the code to evaluate that **redundant expression** only once.

**DAGs** were popular in the ’70s due to memory constraints. **Hash consing** in LISP and Scheme systems also simplified equality tests.

† In a linear IR, sharing is encoded in the name space of the values.
Implementing Trees

In many contexts, we teach students to create purpose-built trees

- Node and edge are classes
- Nodes of various types and arities
- Allocating nodes of different sizes is an issue for the system (*inefficient but functional*)

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

(See §2.3.2 in Knuth Vol 1 or §B.3 in EaC2e)

Tree implementations often become complex & bloated. Knuth’s mapping trick lets the compiler writer stick with a single format for nodes, which simplifies allocation and helps keep things simple. (Informal survey in early 90s showed about 1KB per line in FORTRAN ASTs!)
Control-Flow Graph

Models the transfers 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
```

Implementations:
See Figures B.3 and B.4 in Appendix B of EaC2e
Control-Flow Graph

Models the transfers 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
```

Each **basic block**, a maximal-length sequence of straight-line code, is represented by a **distinct node**.

Implementations:
See Figures B.3 and B.4 in Appendix B of EaC2e
Control-Flow Graph

Models the transfers 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

![Control-Flow Graph Example](image)

Control-flow transfers (e.g., branches and jumps) are represented as edges between the nodes.

Note that we cannot place the edges until we see their targets, making it difficult to generate a CFG straight out of the parser (see Slide 5).

Implementations:
See Figures B.3 and B.4 in Appendix B of EaC2e
Taxonomy of Intermediate Representations

Three major categories of IR

• **Structural IRs**
  - Graphs and trees
  - Widely used in source-to-source translators
  - Tend to use large amounts of memory

• **Linear IRs**
  - Pseudo-code for an abstract machine
  - Simple compact data structures
  - Can be easy to reorder and rearrange

• **Hybrid IRs**
  - Combinations of graphs and linear code
  - Provide some of the advantages of both structural & linear IRs

**Examples:***
- Trees, directed acyclic graphs, interference graph (register allocation)
- Three address code (**ILOC**), stack machine code (Java bytecode), register-transfer language (**RTL**)
- Control-flow graph, SSA form
Stack Machine Code  (or One-Address Code)

Originally used for stack-based computers, now used as an IR

Example

\[ x - 2 \times y \] becomes

push 2
push y
Multiply
push x
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 is essentially a stack machine code
• It was designed for transmission over slow links (bytecode)
Three-Address Code

Three-address code is modeled on assembly code

- In general, three-address code has the form
  \[ x \leftarrow y \ op \ z \]
  with at most one operator \((op)\) and three names \((x, y, \& z)\)

- Three-address code introduces a new name space

Example

\[ z \leftarrow x \ 2 \ast y \]
becomes

Advantages:

- Resembles many real machines
- Introduces a new set of names
- Compact form
Three-Address Code

Quadruples: A straightforward implementation of three-address code

- Use a table of $k \times 4$ small integers
- Simple record structure
- Instantiate the explicit names
- Easy, albeit slow, to reorder

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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>$O_1$</th>
<th>$O_2$</th>
<th>$O_3$</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**

**Quadruples**

Map opcodes into integers

Other fields interpreted based on position and opcode
Three-Address Code

An updated version of quadruples

• Represent each operation as a 1 x 4 array of integers
  ◆ Types of operands are known contextually from opcode

• Link together the operations in a (doubly) linked list

• Simple record structure

• Easy to insert, delete, & reorder

• For efficiency, may want to build a custom allocator for the operation structures
  ◆ Get them 100 or 1,000 at a time and dole them out with a cheap macro or inline call.

List of Quadruples
Taxonomy of Intermediate Representations

Three major categories of IR

• **Structural IRs**
  - Graphs and trees
  - Widely used in source-to-source translators
  - Tend to use large amounts of memory
  
  **Examples:**
  Trees, directed acyclic graphs, interference graph (register allocation)

• **Linear IRs**
  - Pseudo-code for an abstract machine
  - Simple compact data structures
  - Can be easy to reorder and rearrange

  **Examples:**
  Three address code (ILOC), stack machine code (Java bytecode), register-transfer language (RTL)

• **Hybrid IRs**
  - Combinations of graphs and linear code
  - Provide some of the advantages of both structural & linear IRs

  **Examples:**
  Control-flow graph, SSA form
Hybrid IRs

Control-Flow Graph, Revisited

The control-flow graph is, almost always, a hybrid that combines two distinct representations: a graph for control and a linear IR for operations

- Graph navigation where helpful
- Linear IR for the blocks
- Explicit & obvious control flow
- Explicit ordering for optimization and code generation

Example, Revisited

if (x = y) then
    a ← 2
    b ← 5
else
    a ← 3
    b ← 4
    c ← a * b

Source Code

The Graph

The Individual Blocks

B₁ : If (x = y)
    goto B₁ or B₂

B₂ : a ← 2
    b ← 5

B₃ : a ← 3
    b ← 4

B₄ : c ← a * b

COMP 506, Spring 2019
Static Single-Assignment Form

The Main Idea: each name is defined by exactly one operation

The Problem: at an operation, need one name for each operand

The Solution: introduce artificial functions, called $\phi$-functions to unify names

Original Code

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

Code in SSA-form

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

Strengths of SSA Form

- Sharper analysis $\Rightarrow$ better code
- (sometimes) faster algorithms
- $\phi$-functions mark points where values merge

SSA Form is often interpreted as a graph, with edges running from a definition to a use
Many recent compilers use multiple representations

- gcc uses two levels: high-level (GIMPLE) and low-level (RTL)
- clang uses two levels: llvm code and a non-SSA version in code generation
- Open 64 uses 5 distinct levels of WHIRL
- Repeatedly lower the level of abstraction and optimize
  - Introduce more detail at each lowering
  - Optimizations at different levels target different effects
- Compilers are good at lowering and elaborating
- Strategy allows each optimization to see an appropriate IR
And There is More

A compiler’s IR also includes several tables and maps

• **Symbol table**
  - Maps names (strings) to small integers
  - Stores type and location information with the name
  - Fast access to annotations based on either name or number

• **Constant table**
  - A place to collect all of the code’s literal constants and their types
  - Used to generate a static data area — the “constant pool”
  - Constructed “on-the-fly” as constants are encountered

• **Storage map**
  - Shows where each name is stored
  - Memory locations, virtual registers, or implicit (literal constants in a `loadl`)
  - Typically constructed between parsing the declarations and the executables

We will return to these tables in later lectures
A Compiler’s Symbol Table

- The symbol table has an entry for every name that the compiler knows
  ♦ Variables, labels, procedures, classes, system calls, ...
- The entry for a name has a bunch of associated fields
  ♦ Lexeme, type, storage class, address, length, volatility, ...

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Addr</th>
<th>Length</th>
<th>Storage Class</th>
<th>Volatile</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>int</td>
<td>0</td>
<td>4</td>
<td>local</td>
<td>no</td>
</tr>
<tr>
<td>w</td>
<td>Char *</td>
<td>IN VR 1</td>
<td>4</td>
<td>local</td>
<td>no</td>
</tr>
<tr>
<td>x</td>
<td>char</td>
<td>8</td>
<td>12</td>
<td>static</td>
<td>no</td>
</tr>
<tr>
<td>z</td>
<td>float</td>
<td>4</td>
<td>4</td>
<td>global</td>
<td>yes</td>
</tr>
</tbody>
</table>

The compiler can use a string’s table index as a name for x. These “short” names have uniform length and are easy to compare and test.
Digression (or Rant)

The role of string data in a compiler’s intermediate representation

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 to convert each string to a small integer
  - *In the scanner—pass the lexemes as table indices (!)*
- Represent the string with the integer
- Compare strings as integers
A Compiler’s Symbol Table

In either an Algol-like language (ALL) or an Object-Oriented Language (OOL), the compiler must represent names at multiple scopes

- An ALL has lexical scopes; an OOL, has several kinds of scopes
- A simple and effective way to build scoped tables is to chain symbol tables
- One table per scope
- Lookup fails its way up the chain

“Sheaves of tables” implementation
A Compiler’s Symbol Tables

In an OOL, the compiler must keep multiple tables

Search Order: lexical, class, global

Multiple “sheaves of tables”