Intermediate Representations

The Glue That Holds a Compiler Together

Comp 412
Intermediate Representations

IR is the vehicle that carries information between phases

• **Front end**: produces an IR version of the input program

• **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.

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

Simple 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 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
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
Examples:
3 address code
Stack machine code
Examples:
Control-flow graph
SSA Form
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
Conceptually simple idea

In general, three address code has statements of the form:

\[ x \leftarrow y \text{ op } z \]

with, at most, 3 names (\( x, y, \) & \( z \))

Example:

\[ z \leftarrow x \, 2 \times y \]

becomes

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

Advantages:

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

See Labs 1, 2, 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>Op1</th>
<th>Op2</th>
<th>Op3</th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
<td>1</td>
<td></td>
<td>y</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></td>
<td>x</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

Store opcode & operands as small integers, of course.
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

Think “vector of structures or records”

RISC assembly code

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

Quadruples

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Op1</th>
<th>Op2</th>
<th>Op3</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></td>
<td>x</td>
</tr>
<tr>
<td>sub</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

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>

Implicit names occupy no space

If you squint and look at this sideways, it looks like a tree, with all the pointers ...

Remember, for a long time, 640 KB was a lot of RAM
BEGIN INTEGER K;
 ARRAY A[1:I-J];
 K := 0;
 L: IF I > J
 THEN K := K + A[I-J]*6
 ELSE BEGIN I := I+1; I := I+1; GO TO L END
 END

FIGURE 11.1. An ALGOL Program Segment.

(1) BLOCK
(2) - I, J
(3) BOUNDS 1, (2)
(4) ADEC A
(5) := 0, K
(6) - I, J
(7) BMZ (13), (6)
(8) - I, J
(9) * A[(8)], 6
(10) + K, (9)
(11) := (10), K
(12) BR (18)
(13) + I, 1
(14) := (13), I
(15) + I, 1
(16) := (15), I
(17) BRL L
(18) BLCKEND

FIGURE 11.7. Triples for Program Segment of Figure 11.1.

Two-Address Code

Two-address code allows statements of the form

\[ x \leftarrow x \, \text{op} \, y \]

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

**Example**

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

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

**We write:**

\[ r_1 + r_2 \Rightarrow r_2 \]

**as:**

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

**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

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

\[\text{push } x\]
\[\text{push } 2\]
\[\text{push } y\]
\[\text{multiply}\]
\[\text{subtract}\]

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

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

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

Basic blocks: Maximal length sequences of straightline code
Static Single-Assignment Form

The Fundamental Idea: A Name Space In The IR That Simplifies Analysis

1. Each name in the code is defined exactly once
2. Each use refers to exactly one name

Why do we want these properties?

• Provide a unique name for each value
  – Simplifies many optimizations by eliminating kills (def no longer kills name)
• Expose both the flow of values & the ranges of values
  – Can simplify implementation of both analyses & transformations

Why not use SSA for everything?

• Some compilers, in effect, do use SSA as their primary IR
  – LLVM/CLANG, FLANG
• Some aspects of SSA are not easily implementable in code
  – φ-functions have no direct analog in most ISAs
The Fundamental Idea: A Name Space In The IR That Simplifies Analysis

1. Each name in the code is defined exactly once
2. Each use refers to exactly one name

What is easy?

• Straight-line code is trivial
  – New name at every definition
  – Add a subscript, bump it at definition, rewrite with current subscript

• Splits in the CFG are trivial
  – Each block inherits the name space of its (sole) predecessor

What is hard?

• Joins (or merge points) in the CFG are hard
  – Block can inherit two names for the same original program variable
  – Need a mechanism to reconcile such conflicts
Reconciling the two rules of SSA form

• Introduce $\phi$-functions at join points to create new, merged values
  – $\phi$-functions are glorified parallel copies
  – $\phi$-function takes one argument per incoming CFG edge
  – $\phi$-function selects argument that corresponds to control flow

Original Code

```
x ← ...
y ← ...
while (x < k)
x ← x + 1
y ← y + x
```

Same Code in SSA Form

```
x_0 ← ...
y_0 ← ...
while (x < k)
x_1 ← \phi(x_0, x_2)
y_1 ← \phi(y_0, y_2)
x_2 ← x_1 + 1
y_2 ← y_1 + x_2
```

Strengths of SSA-form

• Sharper analysis
• Faster, cleaner algorithms (sometimes)
Combination of IRs

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

• This hybrid IR has 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 abstraction in the IR
  - Each IR 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
Name Spaces

Code is only half the problem

• Names are a fundamental abstraction in most programming languages
• Manageable name spaces are part of the “magic” that allows us to build real systems
  – You can write code without concern that some name was already used
  – New “scopes” create a clean slate for naming

To translate code, the compiler needs:

• Mechanism to map a textual name to a specific entity  “symbol table”
  – Typically, map it back to a specific declaration
• Method to access that entity  “runtime support”
  – Mechanism to generate the entity’s starting virtual address
  – If it is an aggregate, a formula to generate an internal offset
Symbol Table

The concept is clean.
A table to map names into their important attributes.

Compile-time data structure

• Entry for every name ("symbol")
• Associated attributes for each symbol
  – Lexeme, type, address, length, storage class (local, static, global, constant, ...)
• Compiler can use y’s symbol table index as a short name for y in the IR

### Conceptual image of a symbol table

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Addr</th>
<th>Len</th>
<th>Storage Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>int</td>
<td>8</td>
<td>4</td>
<td>local</td>
</tr>
<tr>
<td>w</td>
<td>float</td>
<td>@_w</td>
<td>4</td>
<td>static</td>
</tr>
<tr>
<td>x</td>
<td>char</td>
<td>12</td>
<td>0</td>
<td>static</td>
</tr>
<tr>
<td>z</td>
<td>double</td>
<td>0</td>
<td>8</td>
<td>local</td>
</tr>
</tbody>
</table>
Symbol Tables and Scope Rules

Of course, neither the rules nor the table are that simple

PLs introduce rules to determine the scope associated with each name

• Rules arbitrate clashes of names and declarations (conflicts)
• For each name, determine which declaration defines its properties
• Large programs would be impractical if each name had to be unique

Common case: lexical scoping

• Each “scope” creates a new name space
• Well defined rules for inheriting & obscuring names from outer scopes
• Java, C, C++, and many others follow Algol’s scope rules
  – They differ in which scopes can be nested and which cannot
• The compiler must, of course, keep track of the details
  – And, figure them out during SDT

In lexical scoping, scopes are searched in the order in which they are encountered.
Lexical Scoping in an Algol-like Language

Each scope (procedure, block, structure) creates its own name space

- Any name (almost) can be declared locally
- Local names obscure identical non-local names
- Local names cannot be seen outside the scope
- Scopes are searched in the order that they were encountered

Examples

- Algol and Pascal are the classic examples
  - Nested procedures, often with deep nesting
- Fortran had local, static, and named global scopes
- C added a little (not much) to Fortran
- Scheme has global, procedure-wide, and nested scopes (let)
  - Procedure scope (typically) contains formal parameters
Lexical Scoping in an Algol-like Language

Each scope (procedure, block, structure) creates its own name space
• Any name (almost) can be declared locally
• Local names obscure identical non-local names
• Local names cannot be seen outside the scope
• Scopes are searched in the order that they were encountered

Python is a bit like C, except with different intuitions
• At any point in the code, it has several scopes: builtin, module, and local to a function — functions can be nested
• Assignment to undeclared name creates a symbol in the current scope
• Use before definition in procedure declares a name as global
• Global declaration is needed to assign a global from inside a procedure

Creates a strange set of context-sensitive implicit declarations that defy the intuitions used in programming languages since Algol 60

The intuitions from Python seem to confuse students when they are asked questions about lexical scoping. Remember that Python defies the rules, rather than defining them.
How Does The Compiler Model Scopes?

The compiler needs both compile-time & runtime mechanisms to manage scopes

The compile-time mechanism is a **scoped symbol table**

- Set of individual tables and a **search path**
  - One table per scope
  - Search path depends on syntactic cues and position in the code
- Table must allow cheap & easy addition and deletion of scopes
  - ... and a way to preserve the table for a scope (*debugging*)

The runtime mechanism varies by storage class

- Address generally consists of a **base address** and an **offset**
- Compiler must generate code to compute or access the base address
  - “establish addressability”
- Compiler can then use base + offset to generate an effective address

Lots of important vocabulary on this slide.
How Does The Compiler Model Scopes?

At Compile Time:

The **PL** has some syntactic constructs that start a new scope

- New procedure, new block, class declaration, ...
- Compiler writer adjusts the grammar to have a reduction at the start and end of each block

On block entry:

- Create a new table for the scope & add it to front of search path

On block exit

- Disconnect the table for the old scope & preserve or discard it

We will discuss the runtime support for scopes in the near future
**Lexically-Scoped Symbol Tables**

**High-level idea**
- Create a new table for each scope
- Chain them together into a search path

**“Sheaf of tables” implementation**
- `insert()` may need to create a new table. It always inserts at current level.
- `lookup()` walks chain of tables & returns first occurrence of name
- `delete()` throws away level $p$ table if it is top table in the chain

If the compiler must preserve the table *(for, say, the debugger)*, this picture is even more practical.

Individual tables are hash tables.

Arguably, $O(1)$

This high-level idea can be implemented as shown, or it can be implemented in more space-efficient (albeit complex) ways. (See EaC2e, § B.4)