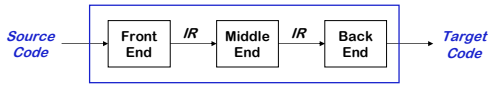
Intermediate Representations

Intermediate Representations (EaC Chaper 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

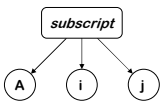
  Examples: Trees, DAGs
- Linear
  - → Pseudo-code for an abstract machine
  - → Level of abstraction varies
  - → Simple, compact data structures
  - → Easier to rearrange

  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:



High level AST:
Good for memory disambiguation

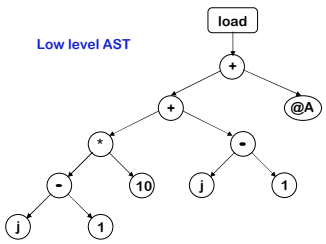
```
loadI 1       => r1
sub   rj, r1  => r2
loadI 10      => r3
mult  r2, r3  => r4
sub   ri, r1  => r5
add   r4, r5  => r6
loadI @A      => r7
Add   r7, r6  => r8
load  r8      => rAij
```

Low level linear code:
Good for address calculation

Level of Abstraction

- Structural *IR*s are usually considered high-level
- Linear *IR*s are usually considered low-level
- Not necessarily true:

Low level AST

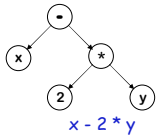



```
loadArray A,i,j
```

High level linear code

Abstract Syntax Tree

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

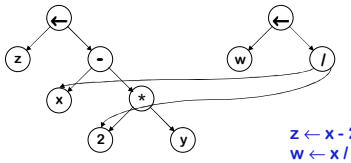


x - 2 * y

- 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 ← x - 2 * y
w ← 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 * 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

Implicit names take up no space, where explicit ones do!

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 ← y *op* z

With 1 operator (*op*) and, at most, 3 names (x, y, z)

Example:

z ← x - 2 * y becomes

t ← 2 * y
z ← 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 * 4 small integers
- Simple record structure
- Easy to reorder
- Explicit names

The original FORTRAN compiler used "quads"

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

RISC assembly code

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

Quadruples

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

stmt array				
(103)	(100)	load	y	
(101)	(101)	loadI	2	
(100)	(102)	mult	(100)	(101)
(102)	(103)	load	x	
(104)	(104)	sub	(103)	(102)

- 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 ϕ-functions to make it work

Original

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

SSA-form

```
      x0 ← ...
      y0 ← ...
      if (x0 > k) goto next
loop: x1 ← ϕ(x0,x2)
      y1 ← ϕ(y0,y2)
      x2 ← x1 + 1
      y2 ← y1 + x2
      if (x2 < k) goto loop
next:    ...
```

Strengths of SSA-form

- Sharper analysis
- ϕ-functions give hints about placement
- (sometimes) faster algorithms

Two Address Code

- Allows statements of the form

  x ← x *op* y

  Has 1 operator (*op*) and, at most, 2 names (x and y)

Example:

z ← x - 2 * y becomes

```
t1 ← 2
t2 ← load y
t2 ← t2 * t1
z  ← load x
z  ← z - t2
```

- Can be very compact

Problems

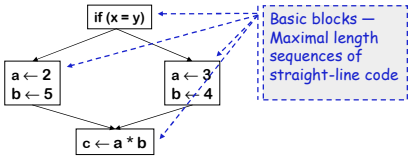
- Machines no longer rely on destructive operations
- Difficult name space
  - → Destructive operations make reuse hard
  - → Good model for machines with destructive ops (PDP-11)

Control-flow Graph (CFG)

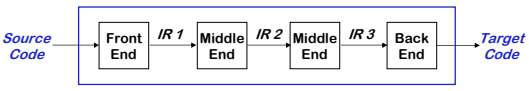
Models the transfer of control in the procedure

- 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 *IR*s 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

`ineg`	[...:i] → [...:-i]	
`iadd`	[...:i1,i2] → [...:i1+i2]	
`isub`	[...:i1,i2] → [...:i1-i2]	
`imul`	[...:i1,i2] → [...:i1*i2]	
`idiv`	[...:i1,i2] → [...:i1/i2]	

Direct instructions

`iinc k a`	[...] → [...]	local[k] ← local[k]+a

Java Byte Codes

Branch instructions

`goto L`	[...] → [...]	branch to L
`ifeq L`	[...:i] → [...]	branch if i = 0
`ifne L`	[...:i] → [...]	branch if i != 0
`ifnull L`	[...:o] → [...]	branch if o = null
`ifnonnull L`	[...:o] → [...]	branch if o != null
`if_icmpeq L`	[...:i1:i2] → [...]	branch if i1 = i2
`if_icmpne L`	[...:i1:i2] → [...]	branch if i1 != i2
`if_icmpgt L`	[...:i1:i2] → [...]	branch if i1 > i2
`if_icmplt L`	[...:i1:i2] → [...]	branch if i1 < i2
`if_acmpeq L`	[...:o1:o2] → [...]	branch if o1 = o2
`if_acmpne L`	[...:o1:o2] → [...]	branch if o1 != o2

Java Byte Codes

Constant loading

`iconst_0`	[...] → [...:0]	
`iconst_1`	[...] → [...:1]	
`aconst_null`	[...] → [...:null]	
`ldc_int i`	[...] → [...:i]	
`ldc_string s`	[...] → [...:str(s)]	

Locals operations

`iload k`	[...] → [...:local[k]]	
`aload k`	[...] → [...:local[k]]	
`istore k`	[...:i] → [...]	local[k]←i
`astore k`	[...:o] → [...]	local[k]←o

Java Byte Codes

Stack operations

`dup`	[...:v] → [...:v,v]
`pop`	[...:v] → [...]
`swap`	[...:v1,v2] → [...:v2,v1]

Functions

`invoke`	[...:args] → [...]	push stack frame, ...
`ireturn`	[...:i] → [...]	ret i, pop stack frame
`areturn`	[...:o] → [...]	ret o, pop stack frame
`return`	[...] → [...]	pop stack frame

Java Byte Code Interpreter

```
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;
        ...
    }
    pc ← new_pc;
}
```