Intermediate representations

Front end produces an intermediate representation (IR) for the program.

Optimizer transforms the code in IR form into an equivalent program that may run more efficiently.

Back end transforms the code in IR form into native code for the target machine.

The IR encodes knowledge that the compiler has derived about the source program.

Intermediate representations

Advantages

- Compiler can make multiple passes over program
- Break the compiler into manageable pieces
- Support multiple languages and architectures using multiple front & back ends
- Enables machine-independent optimization

Desirable properties

- Easy & inexpensive to generate and manipulate
- Contains sufficient information

Examples

- Abstract syntax tree (AST)
- Directed acyclic graph (DAG)
- Control flow graph (CFG)
- Three address code
- Stack code
Intermediate representations

Broadly speaking, IRs fall into three categories:

**Structural**
- structural IRs are graphically oriented
- examples: trees, directed acyclic graphs
- heavily used in source to source translators
- nodes, edges tend to be large

**Linear**
- pseudo-code for some abstract machine
- large variation in level of abstraction
- simple, compact data structures
- easier to rearrange

**Hybrids**
- combination of graphs and linear code
- attempt to take best of each
- examples: control-flow graph

Abstract syntax tree

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

```
-  
  \langle id, x \rangle
  \langle num, 2 \rangle
  \langle id, y \rangle

\text{This represents} \quad x - 2 \ast y \quad \text{.}
```

For ease of manipulation, can use a linearized (operator) form of the tree.

```
x 2 y \ast - \quad \text{in postfix form.}
```
Directed acyclic graph

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

\[
\begin{align*}
\text{id},&x \\
+ \quad \text{sin} \
\end{align*}
\]

\[
\begin{align*}
\text{id},&y \\
\text{num},&2 \\
\times \quad \text{id},&x \\
\text{id},&z \\
\times&t \text{id},&x \\
\text{id},&x \\
\end{align*}
\]

\[
x \leftarrow 2 \times y + \sin(2\times x) \\
z \leftarrow x / 2
\]

Control flow graph

The control flow graph (CFG) models the transfers of control in the procedure.

- nodes in the graph are basic blocks
  maximal-length straight-line blocks of code
- edges in the graph represent control flow
  loops, if-then-else, case, goto

Example

\[
\begin{align*}
\text{if } (x=y) \\
\quad &\text{then } s1 \\
\quad &\text{else } s2 \\
&\text{s3}
\end{align*}
\]

becomes

\[
\begin{align*}
\text{x=y} \\
\quad s1 \\
\quad s2 \\
\quad s3
\end{align*}
\]
Three address code

Three address code generally allow statements of the form:

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

with a single operator and, at most, three names.

Complex expressions like

\[ x - 2 \times y \]

are simplified to

\[ \begin{align*}
  t1 & \leftarrow 2 \times y \\
  t2 & \leftarrow x - t1
\end{align*} \]

Advantages

- compact form (direct naming)
- names for intermediate values

Register transfer language (RTL)

- only load/store instructions access memory
- all other operands are registers
- version of three address code for RISC

Three address code

Typical statement types

1. assignments — \[ x \leftarrow y \text{ op } z \]
2. assignments — \[ x \leftarrow \text{ op } y \]
3. assignments — \[ x \leftarrow y[i] \]
4. assignments — \[ x \leftarrow y \]
5. branches — \[ \text{goto } L \]
6. conditional branches — \[ \text{if } x \text{ relop } y \text{ goto } L \]
7. procedure calls — \[ \text{param } x \text{ and call } p \]
8. address and pointer assignments

Can represent three address code using quadruples

\[
\begin{array}{|c|c|c|c|}
\hline
\text{x - 2 * y} & \text{load} & t1 & y \\
(1) & \text{loadi} & t2 & 2 \\
(2) & \text{mult} & t3 & t2 & t1 \\
(3) & \text{load} & t4 & x \\
(4) & \text{sub} & t5 & t4 & t2 \\
(5) \\
\hline
\end{array}
\]
Stack machine code

Can simplify IR by assuming implicit stack

Example

\[ z = x - 2 \times y \]

becomes

```
push x
push 2
push y
multiply
subtract
store z
```

Advantages

- compact form
- introduced names are implicit, not explicit
- simple to generate and execute code

Disadvantages

- processors operate on registers, not stacks
- difficult to reuse values on stack

Intermediate representations

But this isn’t the whole story

Symbol table:

- identifiers, procedures
- size, type, location
- lexical nesting depth

Constant table:

- representation, type
- storage class, offset(s)

Storage map:

- 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 \[ \ldots i \rightarrow \ldots: -i \]
- iadd \[ \ldots i1,i2 \rightarrow \ldots: i1+i2 \]
- isub \[ \ldots i1,i2 \rightarrow \ldots: i1-i2 \]
- imul \[ \ldots i1,i2 \rightarrow \ldots: i1*i2 \]
- idiv \[ \ldots i1,i2 \rightarrow \ldots: i1/i2 \]

Direct instructions

- iinc \( k \ a \) \[ \ldots \rightarrow \ldots \] \[ local[k] \leftarrow local[k]+a \]

Branch instructions

- goto \( L \) \[ \ldots \rightarrow \ldots \] \[ branch to \( L \) \]
- ifeq \( L \) \[ \ldots i \rightarrow \ldots \] \[ branch if \( i = 0 \) \]
- ifne \( L \) \[ \ldots i \rightarrow \ldots \] \[ branch if \( i \neq 0 \) \]
- ifnull \( L \) \[ \ldots o \rightarrow \ldots \] \[ branch if \( o = \text{null} \) \]
- ifnonnull \( L \) \[ \ldots o \rightarrow \ldots \] \[ branch if \( o \neq \text{null} \) \]
- if_icmpeq \( L \) \[ \ldots i1,i2 \rightarrow \ldots \] \[ branch if \( i1 = i2 \) \]
- if_icmpeq \( L \) \[ \ldots i1,i2 \rightarrow \ldots \] \[ branch if \( i1 = i2 \) \]
- if_icmpne \( L \) \[ \ldots i1,i2 \rightarrow \ldots \] \[ branch if \( i1 \neq i2 \) \]
- if_icmplt \( L \) \[ \ldots i1,i2 \rightarrow \ldots \] \[ branch if \( i1 < i2 \) \]
- if_acmpeq \( L \) \[ \ldots o1,o2 \rightarrow \ldots \] \[ branch if \( o1 = o2 \) \]
- if_acmpne \( L \) \[ \ldots o1,o2 \rightarrow \ldots \] \[ branch if \( o1 \neq o2 \) \]

Java byte codes

Constant loading

- iconst \( 0 \) \[ \ldots \rightarrow \ldots 0 \]
- iconst \( 1 \) \[ \ldots \rightarrow \ldots 1 \]
- aconst_null \[ \ldots \rightarrow \ldots \text{null} \]
- ldc_int \( i \) \[ \ldots \rightarrow \ldots i \]
- ldc_string \( s \) \[ \ldots \rightarrow \ldots \text{str}(s) \]

Locals operations

- iload \( k \) \[ \ldots \rightarrow \ldots \text{local}[k] \]
- aload \( k \) \[ \ldots \rightarrow \ldots \text{local}[k] \]
- istore \( k \) \[ \ldots i \rightarrow \ldots \] \[ local[k] \leftarrow i \]
- astore \( k \) \[ \ldots o \rightarrow \ldots \] \[ local[k] \leftarrow o \]

Stack operations

- dup \[ \ldots v \rightarrow \ldots \text{v},v \]
- pop \[ \ldots v \rightarrow \ldots \]
- swap \[ \ldots v1,v2 \rightarrow \ldots \text{v2},v1 \]

Functions

- invoke \[ \ldots \text{args} \rightarrow \ldots \] \[ push \text{stack frame}, \ldots \]
- ireturn \[ \ldots i \rightarrow \ldots \] \[ \text{ret i, pop \text{stack frame}} \]
- areturn \[ \ldots o \rightarrow \ldots \] \[ \text{ret o, pop \text{stack frame}} \]
- return \[ \ldots \rightarrow \ldots \] \[ \text{pop \text{stack frame}} \]
Java bytecode 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;
        ...
    }
    pc ← new_pc;
}
```