Compiling Techniques
Lecture 9: Intermediate Representations
Christophe Dubach
Intermediate Representations

- 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
Important Properties

- 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 IRs

* **Structural**
  * Graphically oriented
  * Heavily used in source-to-source translator
  * 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 optimisations.
- Two different representations of an array reference:
Level of Abstraction

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

- An AST is the procedure’s parse tree with the nodes for most non-terminal nodes removed:

- Can use linearised form of the tree
  - Easier to manipulate than pointers
  - x 2 y * - in postfix form
  - - * 2 y x in prefix form
A directed acyclic graph (DAG) is an AST with a unique node for each value:

- Makes sharing explicit
- Encodes redundancy
Stack Machine Code

- Originally used for stack-based computers, now JVM

Advantages

- Compact form
- Introduced names are implicit, not explicit
- Simple to generate and execute code
- Implicit names take up no space, but explicit ones do!

Example:

\[ x - 2 \times y \rightarrow \text{becomes} \]

push \( x \)
push 2
push \( y \)
multiply
subtract

Useful where code is transmitted over slow communication links (the net)
Example: JVM Bytecode

- 256 possible opcodes
- Instructions fall into a number of broad groups:
  - Load and store (e.g. `aload_0,istore`)
  - Arithmetic and logic (e.g. `ladd,fcmpl`)
  - Type conversion (e.g. `i2b,d2i`)
  - Object creation and manipulation (new,putfield)
  - Operand stack management (e.g. `swap,dup2`)
  - Control transfer (e.g. `ifeq,goto`)
  - Method invocation and return (e.g. `invokespecial,areturn`)
- There are also a few instructions for a number of more specialised tasks such as exception throwing, synchronisation, etc.
Example: JVM Bytecode

- Stack-oriented

- JVM Bytecode:
  0 iload_1
  1 iload_2
  2 iadd
  3 istore_3

  x86 assembly (for comparison):
  add eax, edx
  mov ecx, eax
Java:

```java
outer:
for (int i = 2; i < 1000; i++) {
    for (int j = 2; j < i; j++) {
        if (i % j == 0)
            continue outer;
    }
    System.out.println (i);
}
```
Example: JVM Bytecode

JVM Bytecode

```
outer:
  for (int i = 2; i < 1000; i++) {
    for (int j = 2; j < i; j++) {
      if (i % j == 0)
        continue outer;
    }
    System.out.println (i);
  }
```

```
0:  iconst_2       // 2
1:  istore_1       // i = 2
2:  iload_1        // i
3:  sipush 1000    // 1000
6:  if_icmpge 44   // if (i < 1000)
9:  iconst_2       // 2
10: istore_2       // j = 2
11: iload_2        // j
12: iload_1        // i
13: if_icmpge 31   // j < i
16: iload_1        // i
17: iload_2        // j
18: irem           // i % j
19: ifne 25        // if (i % j)
22: goto 38        // continue
25: iinc 2, 1      // j++
28: goto 11        // end of j loop
31: getstatic #84; // Field java/lang/System.out:Ljava/io/PrintStream;
34: iload_1        // i
35: invokevirtual #85; // Method java/io/PrintStream.println:(I)V
38: iinc 1, 1      // i++
41: goto 2         // end of i loop
44: return
```
Three Address Code

* In general, three address code has statements of the form: \( x \leftarrow y \text{ op } z \)
  With 1 operator (op) and, at most, 3 names (x, y, & z)

\[
a = b * c + b * d;
\]

\[
t1 = b * c;
\]
\[
t2 = b * d;
\]
\[
t3 = t1 + t2;
\]
\[
a = t3
\]

* Advantages:
  * Resembles many machines
  * Introduces a new set of names
  * Compact form
Example: LLVM IR

* def foo(a b) a*a + 2*a*b + b*b;

define double @foo(double %a, double %b) {
  entry:
    %multmp = fmul double %a, %a
    %multmp1 = fmul double 2.000000e+00, %a
    %multmp2 = fmul double %multmp1, %b
    %addtmp = fadd double %multmp, %multmp2
    %multmp3 = fmul double %b, %b
    %addtmp4 = fadd double %addtmp, %multmp3
    ret double %addtmp4
}
Quadruples

- Naïve representation of three address code
  - Table of k * 4 small integers
  - Simple record structure
  - Easy to reorder
  - Explicit names

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

RISC assembly code
```

```
| load  | 1 | 1 |
| loadI | 2 | 2 |
| mult  | 3 | 2 | 1 |
| load  | 4 | X |
| sub   | 5 | 4 | 2 |

Quadruples
```
SSA: Static Single Assignment Form

- The main idea: each name defined exactly once
- Introduce $\phi$-functions to make it work

**Strengths of SSA form**
- Sharper analysis
- $\phi$-functions give hints about placement
- Enable many optimisations:
  - e.g. constant propagation, dead code elimination, strength reduction
Two Address Code

- Allows statements of the form \( x \leftarrow x \ op \ y \)
- Has 1 operator (op) and, at most, 2 names (x and y)

Example:

- \( z \leftarrow x - 2 \ast y \)

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

- 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 Graphs

- Models the transfer of control in the procedure
- Nodes in the graph are basic blocks
  - Can be represented with three address code or any other linear representation
- Edges in the graph represent control flow
Multiple IRs

- Repeatedly lower the level of the intermediate representation
  - Each intermediate representation is suited towards certain optimisations

- Example: the Open64 compiler
  - WHIRL intermediate format
  - Consists of 5 different IRs that are progressively more detailed
Memory 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
  - Constant table
    - Representation, type
    - Storage class, offset
  - Storage map
    - Overall storage layout
    - Overlap information
    - Virtual register assignments