CMSC 430
Introduction to Compilers
Spring 2016

Intermediate Representations
and
Bytecode Formats
Introduction

- Front end — syntax recognition, semantic analysis, produces first AST/IR
- Middle end — transforms IR into equivalent IRs that are more efficient and/or closer to final IR
- Back end — translates final IR into assembly or machine code
Three-address code

• Classic IR used in many compilers (or, at least, compiler textbooks)

• Core statements have one of the following forms
  - $x = y \ op \ z$  binary operation
  - $x = \ op \ y$  unary operation
  - $x = y$  copy statement

• Example:

  $z = x + 2 \ * \ y$;  $t = 2 \ * \ y$

  $z = x + t$

  • Need to introduce *temporarily variables* to hold intermediate computations
  • Notice: closer to machine code
Control Flow in Three-Address Code

• How to represent control flow in IRs?
  - l: statement labeled statement
  - goto l unconditional jump
  - if x rop y goto l conditional jump (rop = relational op)

• Example

```plaintext
if (x + 2 > 5)
  y = 2;
else
  y = 3;
x++;  

```

```plaintext
  t = x + 2
  if t > 5 goto l1
  y = 3
  goto l2
l1: y = 2
l2: x = x + 1
```
Looping in Three-Address Code

• Similar to conditionals

```c
x = 10;
while (x != 0) {
    a = a * 2;
    x++;
}
y = 20;
```

```c
x = 10
l1: if (x == 0) goto l2
    a = a * 2
    x = x + 1
    goto l1
l2: y = 20
```

- The line labeled l1 is called the *loop header*, i.e., it’s the target of the backward branch at the bottom of the loop.
- Notice same code generated for

```c
for (x = 10; x != 0; x++)
    a = a * 2;
y = 20;
```
Basic Blocks

• A basic block is a sequence of three-addr code with
  ▪ (a) no jumps from it except the last statement
  ▪ (b) no jumps into the middle of the basic block

• A control flow graph (CFG) is a graphical representation of the basic blocks of a three-address program
  ▪ Nodes are basic blocks
  ▪ Edges represent jump from one basic block to another
    - Conditional branches identify true/false cases either by convention (e.g., all left branches true, all right branches false) or by labeling edges with true/false condition
  ▪ Compiler may or may not create explicit CFG structure
Example

1. a = 1
2. b = 10
3. c = a + b
4. d = a - b
5. if (d < 10) goto 9
6. e = c + d
7. d = c + d
8. goto 3
9. e = c - d
10. if (e < 5) goto 3
11. a = a + 1

1. a = 1
2. b = 10
3. c = a + b
4. d = a - b
5. d < 10
6. e = c + d
7. d = c + d
9. e = c - d
10. e < 5
11. a = a + 1
Levels of Abstraction

• Key design feature of IRs: what level of abstraction to represent
  ▪ if x rop y goto l  with explicit relation, OR
  ▪ t = x rop y; if t goto l  only booleans in guard
  ▪ Which is preferable, under what circumstances?

• Representation of arrays
  ▪ x = y[z]  high-level, OR
  ▪ t = y + 4*z; x = *t;  low-level (ptr arith)
  ▪ Which is preferable, under what circumstances?
Levels of Abstraction (cont’d)

• Function calls?
  ▪ Should there be a function call instruction, or should the calling convention be made explicit?
    - Former is easier to work with, latter may enable some low-level optimizations, e.g., passing parameters in registers

• Virtual method dispatch?
  ▪ Same as above

• Object construction
  ▪ Distinguished “new” call that invokes constructor, or separate object allocation and initialization?
Virtual Machines

• An IR has a semantics
• Can interpret it using a virtual machine
  ▪ Java virtual machine
  ▪ Dalvik virtual machine
  ▪ Lua virtual machine
  ▪ “Virtual” just means implemented in software, rather than hardware, but even hardware uses some interpretation
    - E.g., x86 processor has complex instruction set that’s internally interpreted into much simpler form

• Tradeoffs?
Java Virtual Machine (JVM)

- JVM memory model
  - Stack (function call frames, with local variables)
  - Heap (dynamically allocated memory, garbage collected)
  - Constants

- Bytecode files contain
  - Constant pool (shared constant data)
  - Set of classes with fields and methods
    - Methods contain instructions in Java bytecode language
    - Use javap -c to disassemble Java programs so you can look at their bytecode
JVM Semantics

- Documented in the form of a 500 page, English language book
  - [http://java.sun.com/docs/books/jvms/](http://java.sun.com/docs/books/jvms/)
- Many concerns
  - Binary format of bytecode files
    - Including constant pool
  - Description of execution model (running individual instructions)
  - Java bytecode verifier
  - Thread model
JVM Design Goals

• Type- and memory-safe language
  ▪ Mobile code—need safety and security

• Small file size
  ▪ Constant pool to share constants
  ▪ Each instruction is a byte (only 256 possible instructions)

• Good performance

• Good match to Java source code
JVM Execution Model

• From the JVM book:
  ▪ Virtual Machine Start-up
  ▪ Loading
  ▪ Linking: Verification, Preparation, and Resolution
  ▪ Initialization
  ▪ Detailed Initialization Procedure
  ▪ Creation of New Class Instances
  ▪ Finalization of Class Instances
  ▪ Unloading of Classes and Interfaces
  ▪ Virtual Machine Exit
JVM Instruction Set

- **Stack-based language**
  - All instructions take operands from the stack

- **Categories of instructions**
  - 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`)

Example

```java
class A {
    public static void main(void) {
        System.out.println("Hello, world!");
    }
}
```

• Try compiling with javac, look at result using javap -c

• Things to look for:
  ▪ Various instructions; references to classes, methods, and fields; exceptions; type information

• Things to think about:
  ▪ File size really compact (Java → J)? Mapping onto machine instructions; performance; amount of abstraction in instructions
Dalvik Virtual Machine

• Alternative target for Java
• Developed by Google for Android phones
  ▪ Register-, rather than stack-, based
  ▪ Designed to be even more compact
• .dex (Dalvik) files are part of apk’s that are installed on phones (apks are zip files, essentially)
  ▪ All classes must be joined together in one big .dex file, contrast with Java where each class separate
  ▪ .dex produced from .class files
The DVM reads instructions by units of two bytes. This increased code size has limited impact on performance, than in Java, but have a 35% larger code size (bytes) [17]. Applications require fewer instructions. Applications encoded in Dalvik bytecode have on average 30% fewer instructions. 

The nature of the opcodes is very different than that of Java: 218 opcodes vs. 200, respectively. The DVM is register-based, whereas Java has tens of opcodes dedicated to pushing and pulling elements between the stack and local variables. Moreover, as illustrated in Figure 3, Dalvik instructions tend to manipulate registers, rather than accessing operand stack sizes) and to class and instance variables. The data element contains the method code executed by the target VM, as well as other information related to methods definitions, and the data segment. A constant pool describes, for example, Java has tens of opcodes dedicated to pushing elements to the operand stack; the Dalvik bytecode shown in Figure 3(b) pushes onto the operand stack using the iload opcode. By comparison, the Dalvik bytecode shown in Figure 3(c) simply references the registers directly.

Figure 3: Register vs. Stack Opcodes

- Many .class files ⇒ one .dex file
- Enables more sharing

Dalvik is Register-Based

```java
public int add(int a, int b)
{
    return a + b;
}
```

(a) Source Code

```java
public int add(int a, int b)
{
    return a + b;
}
```

(b) Java (stack) bytecode

```java
public int add(int, int)
0:  iload_1
1:  iload_2
2:  iadd
3:  ireturn
```

(c) Dalvik (register) bytecode

```java
public int add(int, int)
0:  add-int v0,v2,v3
2:  return v0
```
The Dalvik constant pool contains information similar to that in the Java constant pool, but with significant differences. Each constant pool entry in Dalvik typically points to a data structure, whereas in Java, constant pool entries are more directly associated with methods and classes. Dalvik uses a different format for representing these structures, which can lead to more compact bytecode and potentially better performance.

In Dalvik, constant pool entries can include:
- **CONSTANT-Class_info**: Contains the name and type index of the class.
- **CONSTANT-NameAndType_info**: Holds the name and type index of a method or field.
- **CONSTANT-Utf8_info**: Stores string data, used in method and field names or types.

To translate Dalvik bytecode to Java bytecode, an analysis and translation tool (ded) is used. This tool performs several steps:
1. Identifies the constants required by a class.
2. Populates the Java constant pool with these constants.

The two constant pool entries differ significantly from each other. The Dalvik format avoids overhead by storing more information in each entry, whereas the Java format keeps the pool entries lean by using more references to reduce memory usage. This distinction is crucial for understanding the performance differences between Dalvik and Java virtual machines.
Dalvik Levels of Indirection

[Diagram showing the structure of Dalvik constant pool entries for method references, with arrows indicating the relationships between constant pool items and the types of data they represent.]

The diagram illustrates the structure of Dalvik constant pool entries for method references, showing how different types of data (such as class indexes, method indexes, and string data) are organized and interconnected. Each box represents a constant pool entry type, with arrows indicating the flow of information and the relationships between these types. The diagram is designed to help understand the complexity and layout of Dalvik bytecode instructions, as well as the process of method code retargeting.
Discussion

• Why did Google invent its own VM?
  - Licensing fees? (C.f. current lawsuit between Oracle and Google)
  - Performance?
  - Code size?
  - Anything else?
Just-in-time Compilation (JIT)

• Virtual machine that compiles some bytecode all the way to machine code for improved performance
  ▪ Begin interpreting IR
  ▪ Find performance critical sections
  ▪ Compile those to native code
  ▪ Jump to native code for those regions

• Tradeoffs?
  ▪ Compilation time becomes part of execution time
Trace-Based JIT

• Recently popular idea for Javascript interpreters
  - JS hard to compile efficiently, because of large distance between its semantics and machine semantics
    - Many unknowns sabotage optimizations, e.g., in e.m(...), what method will be called?

• Idea: find a critical (often used) trace of a section of the program’s execution, and compile that
  - Jump into the compiled code when hit beginning of trace
  - Need to be able to back out in case conditions for taking trace are not actually met