Chapter 15::
Run-time Program Management

Programming Language Pragmatics

Michael L. Scott

Copyright © 2009 Elsevier

Introduction

Run-time system:

Refers to the set of libraries on which the language implementation depends for correct operation

- Some parts of the run-time, obtain all the information they need from subroutine arguments

- Others require more extensive knowledge of the compiler or the generated program.

Copyright © 2009 Elsevier

ELSEVIER

Run-time system

- Simple cases

  This knowledge is a set of conventions that the compiler and runtime both respect.

- More complex cases

  The compiler generates program-specific metadata that the runtime must inspect to do its job.

Copyright © 2009 Elsevier

Introduction

Many examples of compiler/runtime integration have been discussed:

- Garbage Collection (*Section 7.7.3*)
- Variable Numbers of Arguments (*Section 8.3.3*)
- Exception Handling (*Section 8.5*)
- Event Handling (*Section 8.7*)
- Coroutine & Thread Implementation (*8.6* & *12.2.4*)
- Remote Procedure Call (*Section 12.5.4*)
- Transactional Memory (*Section 12.4.4*)
- Dynamic Linking (*Section 14.7*)

Copyright © 2009 Elsevier

Introduction

The length and complexity of the list above generally means that the compiler and the run-time system must be developed together.

- Some languages have very small run-time systems: most user level code required to execute a given source program is either generated directly by the compiler or contained in language-independent libraries.
- Other languages have extensive run-time systems. e.g. C# is heavily dependent on a run-time system defined by the Common Language Infrastructure standard which depends on data generated by the compiler

Copyright © 2009 Elsevier

Virtual Machines

A virtual machine (VM) provides a complete programming environment

- Its application programming interface (API) provides all requirements for execution of programs that run above it

- Is term used for environments whose level of abstraction is comparable to that of a computer implemented in hardware

Copyright © 2009 Elsevier

Virtual Machines

Virtual machines tend to be characterized as either:

- System VM

  Faithfully emulates all the hardware facilities needed to run a standard OS, including both privileged and unprivileged instructions, memory-mapped I/O, virtual memory, and interrupt facilities. (sometimes referred to as virtual machine monitors (VMMs))

- Process VM

  Provides the environment needed by a single user-level process: the unprivileged subset of the instruction set and a library-level interface to I/O and other services.

Copyright © 2009 Elsevier

Virtual Machines

The Java Virtual Machine (JVM)

- Started as a development of the language Java in 1990–91 at Sun Microsystems 1st public release of Java occurred in 1995

- Code in the JVM was entirely interpreted

- JIT compiler added in 1998, with release of Java 2

Copyright © 2009 Elsevier

JVM Architecture Summary

- Interface provided by JVM designed as target for a Java compiler
- Provides direct support for all the built-in and reference types defined by the Java language
- Enforces both definite assignment (Section 6.1.3) & type safety.
- Includes built-in support for many of Java's language features & standard library packages, including exceptions, threads, garbage collection, reflection, dynamic loading, & security
- Usually Java byte code (JBC) is produced from Java source- however, compilers targeting the JVM exist for many languages, including Ruby, JavaScript, Python, Scheme, C, Ada, Cobol, & others

Copyright © 2009 Elsevier

Virtual Machines

JVM Storage Management Storage allocation mechanisms in the JVM mirror those of the Java language:

- Global constant pool
- Set of registers
- Stack for each thread
- Method area to hold executable byte code
- Heap for dynamically allocated objects

Copyright © 2009 Elsevier

Virtual Machines

A trivial "Hello, world" program (Example 15.2):

```
class Hello {
    public static void main(String args[]) {
        System.out.println("Hello, world!");
    }
};
```

Copyright © 2009 Elsevier

Virtual Machines

```
const #1 = Method   #6.#15;         //  java/lang/Object."<init>":()V
const #2 = Field    #16.#17;        //  java/lang/System.out:Ljava/io/PrintStream;
const #3 = String   #18;            //  Hello, world!
const #4 = Method   #19.#20;        //  java/io/PrintStream.println:(Ljava/lang/String;)V
const #5 = class    #21;            //  Hello
const #6 = class    #22;            //  java/lang/Object
const #7 = Asciz    <init>;
const #8 = Asciz    ()V;
const #9 = Asciz    Code;
const #10 = Asciz   LineNumberTable;
const #11 = Asciz   main;
const #12 = Asciz   ([Ljava/lang/String;)V;
const #13 = Asciz   SourceFile;
const #14 = Asciz   Hello.java;
const #15 = NameAndType #7:#8;      //  "<init>":()V
const #16 = class   #23;            //  java/lang/System
const #17 = NameAndType #24:#25;    //  out:Ljava/io/PrintStream;
const #18 = Asciz   Hello, world!;
const #19 = class   #26;            //  java/io/PrintStream
const #20 = NameAndType #27:#28;    //  println:(Ljava/lang/String;)V
const #21 = Asciz   Hello;
const #22 = Asciz   java/lang/Object;
const #23 = Asciz   java/lang/System;
const #24 = Asciz   out;
const #25 = Asciz   Ljava/io/PrintStream;;
const #26 = Asciz   java/io/PrintStream;
const #27 = Asciz   println;
const #28 = Asciz   (Ljava/lang/String;)V;
```

Figure 15.1 Content of the JVM constant pool for the program in Example 15.2. The "Asciz" entries (zero-terminated ASCII) contain null-terminated character-string names. Most other entries pair an indication of the kind of constant with a reference to one or more additional entries. This output was produced by Sun's `javap` tool.

Copyright © 2009 Elsevier

JVM Class Files

Class file is stored as a stream of bytes

- Typically, real file provided by the operating system, could just as easily be a record in a database

Multiple class files may be combined into a Java archive (.jar) file

Copyright © 2009 Elsevier

JVM Byte Code (JBC)

Stack oriented, operands and results of arithmetic and logic instructions are kept in the operand stack of the current method frame, rather than in registers

Instruction set, version 2 categories:

- load/store
- arithmetic
- type conversion
- object management
- operand stack management
- control transfer
- method calls
- exceptions & monitors

Copyright © 2009 Elsevier

JVM byte code for a list insert (Example 15.3):

```
public class LLset {
    node head;
    class node {
        int val;
        node next;
    };
    public LLset() {              // constructor
        head = new node();        // head node contains no real data
        head.next = null;
    }
    ...
}
```

Copyright © 2009 Elsevier

Figure 15.2 Java source and byte code for a list insertion method (Example 15.3):

```
public void insert(int v) {
    node n = head;

    while (n.next != null
            && n.next.val < v) {




        n = n.next;


    }
    if (n.next == null
        || n.next.val > v) {






        node t = new node();




        t.val = v;


        t.next = n.next;



        n.next = t;


    } // else v already in set
}
```

```
Code:
 Stack=3, Locals=4, Args_size=2
 0:   aload_0             // this
 1:   getfield      #4; //Field head:LLLset$node;
 4:   astore_2
 5:   aload_2             // n
 6:   getfield      #5; //Field LLset$node.next:LLLset$node;
 9:   ifnull        31  // conditional branch
 12:  aload_2
 13:  getfield      #5; //Field LLset$node.next:LLLset$node;
 16:  getfield      #6; //Field LLset$node.val:I
 19:  iload_1             // v
 20:  if_icmpge     31
 23:  aload_2
 24:  getfield      #5; //Field LLset$node.next:LLLset$node;
 27:  astore_2
 28:  goto          5
 31:  aload_2
 32:  getfield      #5; //Field LLset$node.next:LLLset$node;
 35:  ifnull        49
 38:  aload_2
 39:  getfield      #5; //Field LLset$node.next:LLLset$node;
 42:  getfield      #6; //Field LLset$node.val:I
 45:  iload_1
 46:  if_icmple     76
 49:  new           #2; //class LLset$node
 52:  dup
 53:  aload_0
 54:  invokespecial #3; //Method LLset$node."<init>":(LLLset;)V
 57:  astore_3
 58:  aload_3             // t
 59:  iload_1
 60:  putfield      #6; //Field LLset$node.val:I
 63:  aload_3
 64:  aload_2
 65:  getfield      #5; //Field LLset$node.next:LLLset$node;
 68:  putfield      #5; //Field LLset$node.next:LLLset$node;
 71:  aload_2
 72:  aload_3
 73:  putfield      #5; //Field LLset$node.next:LLLset$node;
 76:  return
```

Figure 15.2 **Java source and byte code for a list insertion method.** Output on the right was produced by Sun's `javac` (compiler) and `javap` (disassembler) tools, with additional comments inserted by hand.

Copyright © 2009 Elsevier

Common Language Infrastructure (CLI)

- Began at Microsoft Corporation in the late 1990s

- Need for interoperability among programming languages running on Windows platforms

- For .NET, a specification for CLI virtual machine was
standardized by ECMA in 2001

Copyright © 2009 Elsevier

Architecture and Comparison to the JVM

Similarities between CLI and JVM:

- Both systems define a multithreaded, stack-based virtual machine, with built-in support for garbage collection, exceptions, virtual method dispatch, and mix-in inheritance.

- Both represent programs using a platform-independent, self-descriptive, byte code notation.

- For languages like C#, the CLI provides all the safety of the JVM, including definite assignment, strong typing, and protection against overflow or underflow of the operand stack.

Copyright © 2009 Elsevier

Architecture and Comparison to the JVM

Contrasts between CLI and JVM:

- Richer type system for CLI

- Richer calling mechanisms in CLI

- Unsafe code made explicit in CLI

- Miscellaneous CLI support

Copyright © 2009 Elsevier

The Common Type System

Built-in Types

- Integers

    in 8, 16, 32, and 64 bit lengths, both signed and unsigned

- “Native” integers

    supported by hardware, again both signed and unsigned

- IEEE floating-point

    both single and double precision Object references and “managed” pointers

Copyright © 2009 Elsevier

The Common Type System

Constructed Types

- Dynamically allocated instances of class, interface, array, and delegate types
- Methods – function types
- Properties – getters and setters for objects
- Events – lists of delegates, associated with an object
- Value types – records (structures), unions, & enumerations
- Boxed value types – values embedded in dynamically allocated object
- Function pointers – references to static functions
- Typed references – pointers bundled, with type descriptor
- Unmanaged pointers – as in C, can point to just about anything, & support pointer arithmetic

Copyright © 2009 Elsevier

The Common Language Specification

- Defines subset of CTS that most languages can accommodate
- Omits several of the types provided by the CTS
- Imposes restrictions on the use of other types; establishes naming conventions, limits the use of overloading, and defines the operators and conversions that programs can assume are supported on built-in types
- None of these restrictions applies to program components that operate only within a given language

Copyright © 2009 Elsevier

Metadata and Assemblies

- Portable Executable (PE) assemblies are the rough equivalent of Java .jar files
- Contain the code for a collection of CLI classes based on the Common Object File Format (COFF), originally developed for AT&T's System V Unix.

The Common Intermediate Language (CIL)

- Version 4 of ECMA standard defines approximately 250 instructions
- CIL bears a strong resemblance to JBC
- Any differences stem from the assumption that CIL will always be JIT-compiled

Copyright © 2009 Elsevier

Just-in-Time (JIT) and Dynamic Compilation

- JIT system compiles programs immediately prior to execution, can add significant delay to program start-up time
- Cost of JIT compilation is typically lessened by the existence of an earlier source-to-byte-code compilere.g. Java byte code (JBC)

Copyright © 2009 Elsevier

Late Binding of Machine Code

Dynamic Compilation

- In some cases JIT compilation must be delayed, either because:
    - Source or byte code was not created or discovered until run time
    - Perform optimizations that depend on information gathered during execution
- Common Lisp, the language is typically compiled, but a program can extend itself at run time

Copyright © 2009 Elsevier

ELSEVIER

Late Binding of Machine Code

Binary Translation

- Recompilation of object code
- Allows already-compiled programs to be run on a machine with a different instruction set architecture

e.g. Apple's Rosetta system, which allows programs compiled for older PowerPC-based Macintosh computers to run on newer x86-based Macs

Copyright © 2009 Elsevier

Binary Translation

- Principal challenge is loss of information in the original source-to-object code translation
  - Object code typically lacks both type information and clearly-delineated subroutines and control-flow constructs of source code and byte code
  - Yet most of this information appears in compiler's symbol table

Copyright © 2009 Elsevier

Binary Translation

- Typical binary translator reads an object file & reconstructs control flow graph (cf. Section 14.1.1)
  - Task complicated by lack of explicit information about basic blocks.
  - While branches (the ends of basic blocks) are easy to identify, beginnings are more difficult: since branch targets are sometimes computed at run time or looked up in dispatch tables or virtual function tables

Copyright © 2009 Elsevier

Late Binding of Machine Code

Binary Rewriting

Technique to modify existing executable code can be used for:

- Profiling (insert instrumentation of some kind)
- Simulate new architectures
- Evaluate the coverage of test suites- implement model checking for parallel programs, a process that exposes race conditions
- “Audit” the quality of a compiler’s optimizations- insert dynamic semantic checks into a program
- “Sandbox” untrusted code, permits an safe execution in same address space as application.

Copyright © 2009 Elsevier

ELSEVIER

Mobile Code and Sandboxing

- Portability is one of the principal motivations for late binding of machine code.
- Code compiled for one machine architecture or operating system cannot generally be run on another due to dependencies.
- For code to be mobile, it must be executed in some sort of sandbox.
- Sandbox mechanisms lie at the boundary between language implementation and operating systems.

Copyright © 2009 Elsevier

Inspection/Introspection

Reflection

- Load already-compiled program, to use reflection tools for querying symbol table information created by compiler

- E.g. reflection useful when printing diagnostics

Copyright © 2009 Elsevier

Inspection/Introspection

Reflection

- C#’s reflection API is similar to that of Java:
  - System.Type is analogous to java.lang.Class
  - System. Reflection is analogous to java.lang.reflect
- All of the major scripting languages (Perl, PHP, Tcl, Python, Ruby, JavaScript) provide extensive reflection mechanisms
- Principal difference between reflection in Java or C# and in scripting languages (Lisp) is the later is dynamically typed

Copyright © 2009 Elsevier

Symbolic Debugging

- Built into most programming language interpreters, virtual machines, and integrated program development environments
- Also available as stand-alone tools e.g. GNU's gdb
- Adjective symbolic refers to a debugger's understanding of high level language syntax

Copyright © 2009 Elsevier

Symbolic Debugging

Debugger allows the user to perform two main kinds of operations:

- Breakpoint: specifies that execution should stop if it reaches a particular location in the source code.
- Watchpoint: specifies that execution should stop if a particular variable is read or written.

Copyright © 2009 Elsevier

Inspection/Introspection

Symbolic Debugging

- Both data and control operations also depend on the ability to manipulate a program from outside:
  - To stop and start it, and to read and write its data.
- Debugger control can be implemented:
  - Interpreters
  - Dynamic binary rewriting- support from operating system(compiled programs only)
- Some processors provide hardware support

Copyright © 2009 Elsevier

ELSEVIER