CSE 505
Lecture #26
December 5, 2012

Java Virtual Machine (JVM)

Java

\[\text{Java Compiler} \rightarrow \text{JVM (}.class \text{ files)} \rightarrow \text{Machine Independent} \]

\[\text{Java JVM interpreter or JVM Just-in-Time compiler} \]

Pentium (for example)

Java Virtual Machine (cont’d)

External representation
(platform independent)

Internal representation
(implementation dependent)

\[\text{LOAD} \rightarrow \text{.class files} \]

classes  primitive types
objects  arrays  strings
methods

The JVM specification allows IMPLEMENTATION to be dependent on target OS/platform, performance requirements, etc.

JVM Memory Areas

Class files \rightarrow \text{Class Loader Subsystem}

\[\text{Method area} \quad \text{Heap} \quad \text{Java stacks} \quad \text{PC registry} \quad \text{Native method stacks} \]

\[\text{Execution engine} \rightarrow \text{Native Method Interface} \rightarrow \text{Native Method Libraries} \]

JVM is a Stack Machine

JVM instructions
- implicitly take arguments from the stack top
- put their result on the top of the stack

Stack is used to:
- pass arguments to methods
- return a result from a method
- store intermediate results while evaluating expressions
- store local variables
**JVM Stack Frame Specification**

- Pointer to runtime constant pool
- New Call Frame is created by JVM instructions for method invocation, e.g. `invokevirtual`, `invokenonvirtual`, etc.
- The operand stack is initially empty, but grows and shrinks during execution. Also used for expression evaluation.

**Stack Frames (cont’d)**

- Base of Stack
- To runtime constant pool
- JVM instructions store and load (for accessing args and locals) use addresses which are numbers from 0 to #args + #locals - 1
- Top of Stack

**JVM Interpreter Cycle**

```java
repeat {
    byte opcode = fetch an opcode;
    switch (opcode) {
        case opCode1:
            fetch operands for opCode1;
            execute action for opCode1;
            break;
        case opCode2:
            fetch operands for opCode2;
            execute action for opCode2;
            break;
        case ...
    }
}
```

**Typed Instructions**

Different op codes for instructions on integers, floats, arrays, reference types, etc.

Example: different types of “load” instructions

- `iload` integer load
- `lload` long load
- `fload` float load
- `dload` double load
- `aload` reference-type load

**Three Kinds of Operands**

Three kinds of operands:
- from the top of the operand stack
- from the bytes following the opcode
- part of the opcode itself

Example:

<table>
<thead>
<tr>
<th>Assembly code</th>
<th>Binary instruction code layout</th>
</tr>
</thead>
<tbody>
<tr>
<td>iload_0</td>
<td>01 25</td>
</tr>
<tr>
<td>iload_1</td>
<td>02 25</td>
</tr>
<tr>
<td>iload_2</td>
<td>03 25</td>
</tr>
<tr>
<td>iload_3</td>
<td>04 25</td>
</tr>
<tr>
<td>iload n</td>
<td>25 n</td>
</tr>
<tr>
<td>wide iload n</td>
<td>196 21 0</td>
</tr>
</tbody>
</table>

**Accessing the Stack**

- args: indexes 0 .. #args - 1
- locals: indexes #args .. #args + #locals - 1

Instruction examples:

- `iload_1` takes something from the args/locals area and pushes it onto the top of the operand stack.
- `iload_3` pops something from the top of the operand stack and places it in the args/locals area.
Operations on Numbers

Arithmetic
- add: iadd, ladd, fadd, dadd
- subtract: isub, lsub, fsub, dsub
- multiply: imul, lmul, fmul, dmul

Conversion
- i2l, i2f, i2d,
- l2f, l2d, f2d,
- f2l, d2i, ...

Byte Code Example
- .java
  f(int a, int b, int c, int d) {
    return (a + b) *c - d;
  }
- .class
  0: iload_1
  1: iload_2
  2: iadd
  3: iload_3
  4: imul
  5: iload_4
  6: isub
  7: ireturn

Static Methods
public class X {

  public static void main(String[] args) {
    add(1, 2);
  }

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

Factorial Example
int fac(int n) {
  int result = 1;
  for (int i=2; i<n; i++) {
    result = result * i;
  }
  return result;
}

Course Highlights & Remarks
Lambda Calculus

- Getting popular in modern PLs, such as Python, Javascript, Ruby
- Church-Turing thesis: \( \lambda \)-calculus equivalent to Turing Machines.
- \( E ::= V \mid \lambda V. E \mid (E E) \)
- Computation = \( \beta \)-reduction
- Confluence and Church-Rosser properties
- Fixed-point op’r (Y) simulates recursion
- Typed Lambda Calculi

Grammars

Context-free grammars specify the syntax while attribute (or definite-clause) grammars specify the semantics, especially for the purpose of translation.

DCGs = CFGs + attributes + rules

Binding Time

- Early binding favors efficiency.
- Late binding favors flexibility.

Static vs dynamic typing/allocation,
Inheritance vs delegation,
etc.

Advanced Control

Advanced control structures can often be modeled by higher-order constructs.

Translation of iterators.

Lazy evaluation in functional languages.

“Continuations” in Scheme.

Continuations

A continuation = “rest of the computation”. It is an “ipdl”, rather than “rpdl”.

```plaintext
void main() { fork(p1); fork(p2); }
void p1(){ .... yield() .... };
void p2(){ .... yield() .... };
fun fork(p) { call/cc(\lambda K. enq(K) ; p());}
fun yield() { call/cc( \lambda K. enq(K) ; run deq() ); }
```

Strong Typing

Strong Typing means Static Type Checkability.

That is, based on a static analysis of the text of the program.

Contrary to popular belief, strong typing does not mean declaring a type with every variable.
Type Inference

Type Inference in ML confers the benefits of strong typing without explicit type declarations.

Every well-formed ML program (without overloaded operators) has an unambiguous type.

The type system must be suitably designed, as in ML, for such a property to hold.

Types Orthogonal to Paradigms

Strong typing is orthogonal to paradigms.

Functional: Lisp uses dynamic typing, ML uses static typing.

Object-Oriented: Python uses dynamic typing, Java uses static typing.

Polymorphism

Two forms of universal polymorphism: Parametric and Subtype

Universal polymorphism $\forall$ universally quantified type

Example: length: $\forall t. \text{list}(t) \rightarrow \text{int}$

maxdepth: $\forall t. \text{Abs_tree}. t \rightarrow \text{int}$

Type Interfaces

Strictly speaking, an interface should have two parts: signature (operation types) and axioms (operation meaning)

Signature: isempty: $\forall t. \text{stack}(t) \rightarrow \text{bool}$

Axioms: isempty(emptystack) = true

isempty(push(s, x)) = false

Evolution of Interfaces

Early languages, Ada and Modula-2, allowed only one implementation for an interface.

ML allows multiple implementations (structures) for an interface (signature), but they cannot be used interchangeably.

Java allows multiple implementations (classes) for an interface and they can be used interchangeably.

Class Relationships

Inheritance is not the only important class relationship. In many applications, association and aggregation class relationships occur more often.

In object-oriented software engineering, using the well-known “use case” driven approach, Class inheritance is decided at a late design stage, as a means of “code factoring”.
Design Patterns

Design Patterns document well-structured uses of classes and their relationships with the goal of achieving more reusable software designs.

The libraries of Java, C++, etc. incorporate several well-known patterns: adapter, decorator, observer, iterator, etc.

Python

Dynamically Typed Language
Supports Object-Oriented and Functional Style
Generators and List Comprehensions
Interactive Programming Language
Automatic Memory Management (like Java)
Used in scripting and non-scripting contexts

class Tree:
def __init__(self, n):
    self.value = n
    self.left = []
    self.right = []
def insert(self, n):
    if self.value == n:
        return
    if self.value < n:
        if not self.right:
            self.right = Tree(n)
        else:
            self.right.insert(n)
    elif not self.left:
        self.left = Tree(n)
    else:
        self.left.insert(n)
def __iter__(self):
    for x in self.left:
        yield x
    yield self.value
    for x in self.right:
        yield x

Logic Programming

Logic programs demonstrate the principle that logic can serve as programs.

Logic programs are useful in program analysis, language processing, and databases.

Prolog was used in Windows/NT 3.5 to represent device information and perform network configuration

Soundness and Completeness

A logic programming interpreter is sound if it gives correct answers, i.e., no wrong answers.

A logic programming interpreter is complete if it returns all correct answers, i.e., no missing answers.

Constraint Languages

Constraints add a very useful dimension to the logic programming paradigm, and are useful in many applications: scheduling, layouts, combinatorial optimization, etc.

The newer LP languages come with some support for constraints.
Scripting Languages

Originated as special-purpose languages (JCL, awk, sed, ...) but have grown into general purpose languages (Javascript, Perl, ...).

They are generally dynamically-typed and promote rapid program development, often at the expense of program clarity and efficiency.