Performance of Scripting Languages

Nate Nystrom
UTA
Static vs. dynamic typing

• Static typing:
  • great for ensuring interfaces used correctly, enabling optimizations, enforcing security policies, ...
  • tiresome to write, difficult to modify, extend

• Dynamic typing:
  • great for rapid development, extension
  • but, brittle

• My current research: Thorn
  • an OO scripting language for distributed systems
  • supports **gradual typing**:
    • add types gradually as the program grows
    • best of both worlds
Thorn

• My current research: Thorn
  • an OO scripting language for distributed systems
  • supports **gradual typing:**
    • add types gradually as the program grows
    • best of both worlds

• Joint work with colleagues at IBM and Purdue:
  • Bard Bloom, John Field, Jan Vitek, Tobias Wrigstad, and others
Dynamic language implementation

- Decided early on to implement Thorn on the JVM

- How should dynamic languages be implemented on the JVM?

- How well do current dynamic languages perform?
Taxonomy

• Terms often used interchangeably (I will do this)

• No hard and fast definitions, but I’ll try anyway:

  • **Dynamic languages**
    • support run-time code or type extension
      • e.g., eval, dynamic inheritance

  • **Dynamically typed languages**
    • type-check at run-time

  • **Scripting languages**
    • languages used for “scripting” in some domain
Characteristics

• Scripting languages are usually...
  • dynamically typed (or untyped)
  • dynamic (e.g., they provide a read-eval-print loop)
  • interpreted
  • high level

• and are often...
  • domain-specific
## Scripting languages

<table>
<thead>
<tr>
<th>Language</th>
<th>Domain</th>
<th>Abstractions</th>
</tr>
</thead>
<tbody>
<tr>
<td>sh, csh, ...</td>
<td>UNIX</td>
<td>pipes, redirection</td>
</tr>
<tr>
<td>AWK</td>
<td>text files</td>
<td>strings, regexes</td>
</tr>
<tr>
<td>Applescript</td>
<td>Mac applications</td>
<td>application dictionaries</td>
</tr>
<tr>
<td>Javascript</td>
<td>client-side web</td>
<td>DOM</td>
</tr>
<tr>
<td>UnrealScript</td>
<td>3D games</td>
<td>actors, lighting</td>
</tr>
<tr>
<td>ActionScript</td>
<td>Flash</td>
<td>images, movies, sound</td>
</tr>
<tr>
<td>PHP</td>
<td>server-side web</td>
<td>HTML</td>
</tr>
<tr>
<td>Groovy</td>
<td>Java</td>
<td>Java objects, lists, maps</td>
</tr>
<tr>
<td>Perl, Python, Ruby</td>
<td>general purpose</td>
<td>objects, lists, maps</td>
</tr>
</tbody>
</table>
Examples

Java
```java
class hello {
    public static void main(String[] args) {
        System.out.println("hello world");
    }
}
```

Scala
```scala
object hello extends Application {
    Console.println("hello world")
}
```

Ruby
```
puts "hello world"
```

PHP
```
<?php
print "hello world\n";
?>
```

Python
```
print "hello world"
```

Groovy
```
println "hello world"
```

Perl
```
print "hello world\n";
```

Thorn
```
println("hello world");
```
Examples

Java

```java
Map<String, Integer> m = new HashMap<String, Integer>();
m.put("one", 1);
```

Scala

```scala
val m = new HashMap[String, Int]();
m += "one" -> 1;
```

Ruby

```ruby
m = {}
m{"one"} = 1
```

Python

```python
m = {}
m{"one"} = 1
```

Perl

```perl
%m = ();
$m{"one"} = 1;
```

PHP

```php
$m = array();
$m["one"] = 1;
```

Groovy

```groovy
def m = [:]
m["one"] = 1
```

Thorn

```thorn
val m = Map();
m("one") = 1;
```
Performance

- 9 language implementations
  - Ruby, Python, Perl, PHP (C interpreters)
  - JRuby, Jython, Groovy (Java interpreters)
  - Java, Scala (Java compilers)
- 42 programs from the Programming Language Shootout site*
  - All programs small, short running (< 10 min)
- Caveats:
  - Not all programs ported to all languages
  - Sometimes different implementation strategies used
- Setup:
  - Macbook Pro, 2.4GHz Intel Core Duo, 2GB RAM
  - JVM: HotSpot JVM 1.5.0, 512MB heap

*http://shootout.alioth.debian.org
Reading the graphs

C interpreters  Java compilers  Java interpreters

less time or less memory

Java = 1.0
Run times / Java

- Dynamic languages often much slower than Java

- C interpreters: ~2-5x
  - can be 12x faster, 145x slower

- Java interpreters: ~16-43x
  - up to 1200x slower
Run times / Java

Or, to scale:

Saturday, May 8, 2010
C interpreter performance

• Overall 2-5x slower than Java

• Implementation:
  • Ruby, Perl, PHP: AST walking interpreter
  • Python: bytecode (Pycode) interpreter

  • Java/Scala: bytecode interpreter + run-time compilation

• Could improve by adopting same techniques as JVMs (and Self before that)
  • difficult, time-consuming to engineer, maintain
  • not very portable

• Better (perhaps): dynamically compile to bytecode, run on JVM
Scala

- Scala used as a proxy for “best possible” performance of typical scripting language
  - Has many of the same features (e.g., closures, iterators) as Python, Ruby, etc
  - Statically compiled to Java bytecode
  - ~2x slower than Java
Java interpreter performance

- Jython, Groovy implementation:
  - dynamic compilation to Java bytecode
- JRuby:
  - AST interpreter in Java

- JRuby, Jython ~4-8x slower than Ruby, Python

- Overall 16-43x slower than Java
  - Mandelbrot: JRuby 1200x, Groovy 420x slower

- Should be able to approach Scala performance with better implementations
Does it matter?

• Often, no
  • Many scripts short running
  • Many scripts are I/O bound
    • database
    • network
    • other processes

• But, when performance does matter:
  • Often rewrite applications in Java or C
  • Lose benefits of programming in high-level language

• For server-side web applications: scalability matters
  • Want fast startup, low memory usage
Why so slow on the JVM?

- Startup costs
- Object model mismatch
- Duck typing
- High-level language features: iterators, closures
Startup time

Hello, World

C interpreters
• 4-20x faster than Java

Java interpreters
• 5-6x slower than Java

Scala
• 2x Java (more class loading)
Object model

- Dynamic languages permit addition of new fields, methods at run time

```python
class MyClass:
    def __init__(self):
        self.f = 1
    def get(self):
        return self.f
```

Python:

```python
>>> x = MyClass()
>>> x.f
1
>>> x.get()
1
>>> x.g = 'a'
>>> x.g
'a'
```

- Objects implemented as hash tables
  - Slower field access, slower dispatch, slower object instantiation, slower GC
Objects

- Binary tree creation, traversal
  - C interpreters
    - 27-152x Java
  - Java interpreters
    - 63-332x Java

- Object instantiation
  - C interpreters
    - 18-74x Java
  - Java interpreters
    - 35-60x Java
Data structures

- Built-in data structures generally more efficient than objects

- Lists
  - C interpreters
    - fast, except PHP (arrays implemented as hash tables!)
  - Java interpreters
    - 5-14x Java

- Hash tables
  - C interpreters
    - 1.5-3.7x Java
  - Java interpreters
    - 4.3-22.7x Java
Duck typing

- If it looks like a duck...
  - Check if field or method exists at selection time

- Difficult to make method dispatch efficient

- How to handle primitives?

```python
class MyClass:
    def __init__(self):
        self.f = 1
    def get(self, self):
        return self.f

>>> x = 'abc'
>>> x.size()
3
>>> x = MyClass()
>>> x.f
1
```
Primitive types

x = 238932;
x = new Foo();

• If variables are untyped, how to know x is actually an int (or not)?
• Must change representation of integers! (and other primitives)
  • box everything into an object?
  • use two words per value? – type and value
Boxing

• Represent primitives as normal objects
• All variables hold an object (or null)

• Ex: represent an integer with `java.lang.Integer`

\[
a + b
\]

=>

\[
Integer.valueOf(((Integer) a).intValue() + ((Integer) b).intValue())
\]

• Compiler can optimize code to unbox when necessary
Tag bits

- Reserve 1-3 bits in each word to identify primitive values
- Advantage: variable fits in a single word
- Disadvantage: extra overhead smaller range of representable values, pointers

- \(12 = 00001100 \rightarrow 001100[00]\)
- \('\backslash f' = 00001100 \rightarrow 001100[01]\)
- new Foo = 00110000 \rightarrow 001100[11]\)
Tag bit tricks

- Integers: use zero bit pattern so integer \( n \) represented by number \( 4n \)
- adding two integers \( a + b \): just add tagged representation
- multiply: \( a \times b \rightarrow a \times (b >> 2) \)

- Pointers: represent 4-byte aligned pointer to address \( p \) by \( p' = p+3 \)
  - (don’t need to be able to address every byte)
  - \([p+k] = [p'+k-3]\)
  - new Foo = 00110000 \(\rightarrow\) 001100[11]
Primitive types and subtyping

- Java, C#: primitives have no subtyping relationship with other types

- Prevents some code reuse,
  - Must write 9 (10) different versions of generic code
    - Object, boolean, byte, short, char, int, long, float, double, (void)

- Workaround: boxing
  - use java.lang.Integer rather than int
Strings

- Strings, regular expressions
- C interpreters
  - 2.5-10x faster than Java
- Java interpreters
  - 4-5x slower than Java

![Regexmatch time / Java](chart)
Virtual dispatch time

- JRuby:
  - AST interpreter
  - lookup method in hash table
  - most overhead is setting up new stack frame

- Jython:
  - lookup method object in hash table
  - invoke **__call__** method of method object

- Groovy:
  - call using reflection API
Dispatch in Thorn

• Compile Thorn class $C$ to Java class $C$
• If class $C$ has method $m$, create interface $\text{method}\_m$ implemented by $C$
• Cast to interface and invoke
• ~15% slower than Java virtual call

```java
class C {
    def m() = 0;
}
x.m();
```

```java
interface method\_m {
    IObject m();
}

class C implements method\_m {
    IObject m() {
        return new ThornInt(0);
    }
}

((method\_m) x).m();
```
Boxing/unboxing

• Nested loops benchmark:
  • 12-107x slower than Java

• JRuby example:
  
  ```ruby
  for i in 1..n
    x = x + 1
  end
  ```

  * `x` unboxed/reboxed at every iteration of loop
Iterators

• Co-routine style iterators [CLU]

• Each subsequent call to iterator (e.g., `elements`) resumes at previous `yield`

• Efficient implementation of `yield` just adjusts stack pointer, but does not pop `elements` stack frame

• On JVM: save iterator state on heap

```python
def elements(self):
    for x in self.left.elements():
        yield x
    yield self.value
    for x in self.right.elements():
        yield x

for x in tree.elements():
    print x
```
Iterators

- Co-routine style iterators [CLU]

- Each subsequent call to iterator (e.g., `elements`) resumes at previous `yield`

- Efficient implementation of `yield` just adjusts stack pointer, but does not pop `elements` stack frame

- On JVM: save iterator state on heap

```python
def elements(self):
    for x in self.left.elements():
        yield x
    yield self.value
    for x in self.right.elements():
        yield x

for x in tree.elements():
    print x
```

```text
{1..}.elements()
{2..}.elements()
{4..}.elements()
caller
```
Peak memory usage

• C implementations of Perl, Python, Ruby, PHP usually have much smaller footprints than Java

• Results from Language Shootout website

• Reference counting

• No numbers for Java implementations
What’s needed

• Want more control over...
  • memory layout than JVM provides
    • for extending objects with new fields, methods
    • for multiple inheritance
  • method dispatch
    • for multiple inheritance, closures, duck typing
  • call stack
    • for iterators

• Options for how to get it:
  • optimize the JVM for code generated for dynamic languages
  • extend the JVM with new bytecodes for dynamic languages
  • implement a dynamic languages library (with JVM support)
  • roll your own VM for dynamic languages
JVM optimizations

- Object inlining
  - Inline hash tables with constant keys

- Optimize lookup closure in hash table & invoke pattern

- Optimize calls through reflection API

- Closure (anonymous class) elimination

- Reduce JVM and interpreter startup time
  - precompile scripting startup code to bytecode
  - precompile bytecode to native code (see Java0, Quicksilver [Serrano et al. 00], MVM [Czajkowski et al. 2001])

- ... need to profile more
JVM extensions

• JSR 292: invokedynamic
  • invoke a virtual method, type-checking at run-time

• Hot-swapping:
  • method replacement
  • class replacement/extension
    • can add new fields, methods
    • what to do with old instances?
    • can do “class replacement” by replacing factory methods
Dynamic languages VM

• Lower-level object model
  • closer to C level of abstraction, but still portable, type safe
  • primitive types + tuples + records + closures/function ptrs

• Memory layout
  • programmer control over object (record) layout
  • stack allocation
  • extensible objects?

• Extensibility
  • languages-specific instructions?
  • pluggable bytecode verifiers?
LLVM

- Low-level VM project
- IR is at assembly level of abstraction
  - But typed
- Frontend for C
- Used to implement run-time compilation of:
  - code to run on a GPU
  - Photoshop filters
  - ...
Conclusions

• Dynamic languages are great for rapid development

• But, current implementations on the JVM perform terribly
  • mismatch between dynamic code and statically typed Java

• Need better virtual machine support for these languages