Performance of Scripting Languages

Nate Nystrom
IBM Research - 24 August 2007

(inspired by) joint work with:
Bard Bloom, John Field, Martin Hirzel, Igor Peshansky,
Mukund Raghavachari, Jan Vitek

Thorn

Object-oriented scripting language for distributed applications

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

Thorn supports optional typing*:
- add types gradually as the program grows
- best of both worlds

Dynamic language implementation

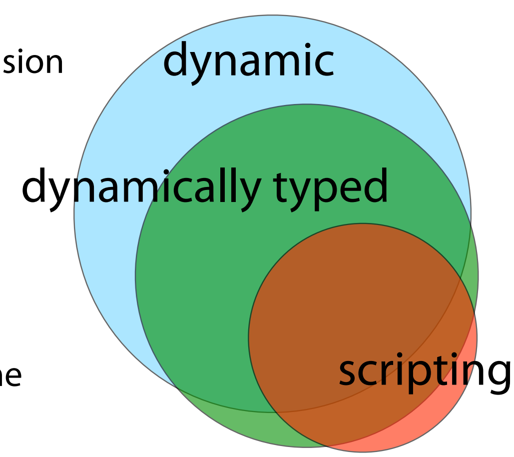
Decided early on to implement Thorn on the JVM

How should dynamic languages be implemented on the JVM?

How well do current dynamic languages perform?

Taxonomy

- Terms often used interchangably (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

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

Examples

Java

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

Scala

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

Ruby

```
puts "hello world"
```

Python

```
print "hello world"
```

Perl

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

PHP

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

Groovy

```
println "hello world"
```

Thorn

```
println(“hello world”);
```


7

Examples

Java

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

Scala

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

Ruby

```
m = {}
m{“one”} = 1
```

Python

```
m = {}
m{“one”} = 1
```

Perl

```
%m = ();
$m{“one”} = 1;
```

PHP

```
$m = array();
$m[“one”] = 1;
```

Groovy

```
def m = [:]
m[“one”] = 1
```

Thorn

```
val m = Map();
m(“one”) = 1;
```


8

Performance

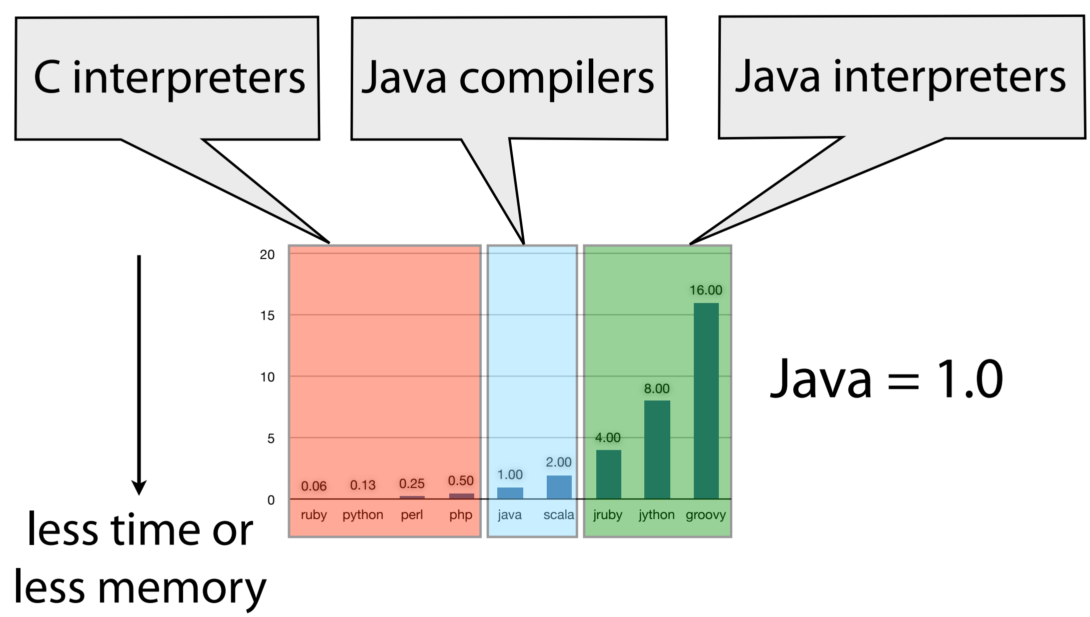
- 9 language implementations
  - Ruby, Python, Perl, PHP (C interpreters)
  - JRuby, Jython, Groovy (Java interpreters)
  - Java, Scala (Java compilers)
- 42 programs from the Programming Language Shootout site*
  - All programs small, short running (< 10 min)
- Caveats:
  - Not all programs ported to 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

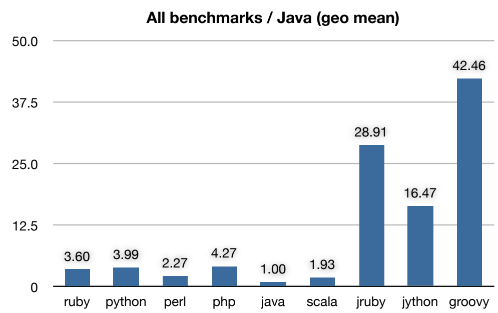
9

Reading the graphs



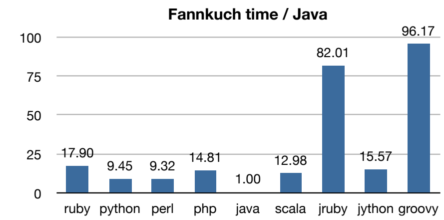
10

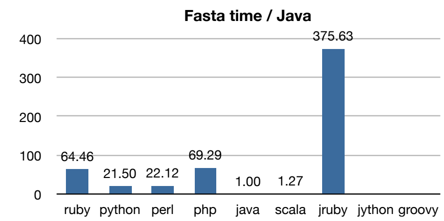
Run times / Java

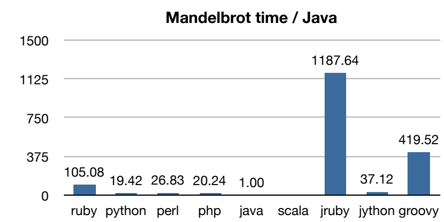


- Dynamic languages often much slower than Java

- C interpreters: ~2-5x
  - can be 12x faster, 145x slower
- Java interpreters: ~16-43x
  - up to 1200x slower







11

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

12

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

13

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

14

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

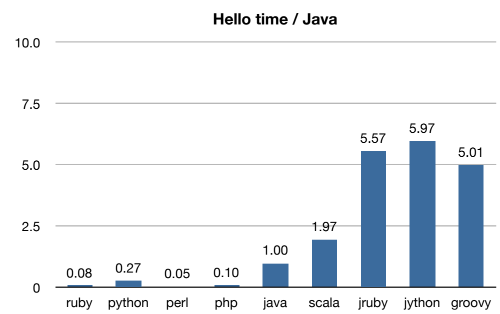
Object model mismatch

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
```

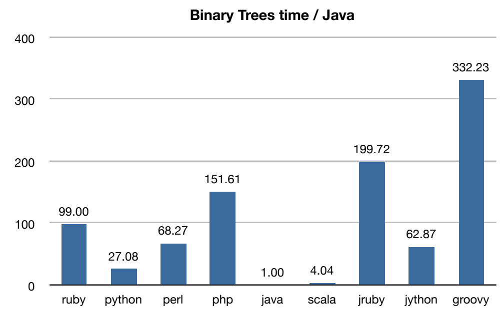
```
>>> x = MyClass()
>>> x.f
1
>>> x.get()
1
>>> x.g = 'a'
>>> x.g
'a'
```

Objects implemented as hash tables

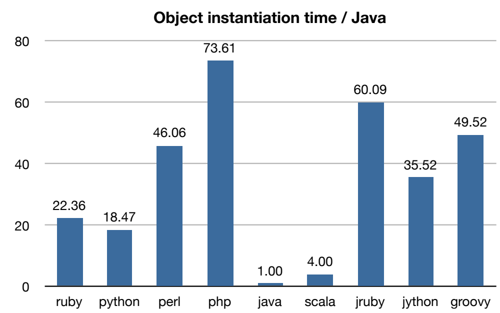
- Slower field access, slower dispatch, slower object instantiation, slower GC

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



Duck typing

If it looks like a duck...

- Check if field or method exists at selection time

Difficult to make method dispatch efficient

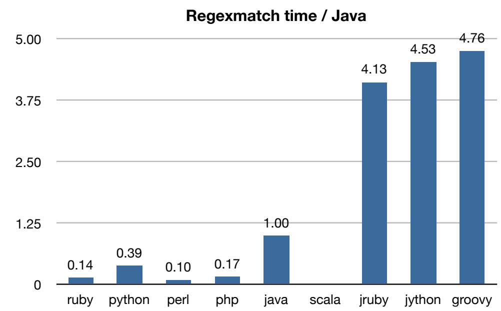
Must box primitive values

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

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

Strings

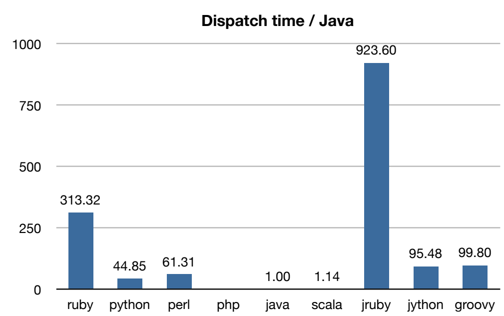
- Strings, regular expressions
- C interpreters
  - 2.5-10x faster than Java
- Java interpreters
  - 4-5x slower than Java



21

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



22

Dispatch in Thorn

Planned implementation:

- Compile Thorn class C to Java class C
- If class C has method m, create interface method$m implemented by C
- Cast to interface and invoke
- ~15% slower than Java virtual call

```
                              interface method$m { IObject m(); }

class C {                     class C implements method$m {
  def m() = 0;       ⇨          IObject m() { return new ThornInt(0); }
}                             }

x.m();                        ((method$m) x).m();
```

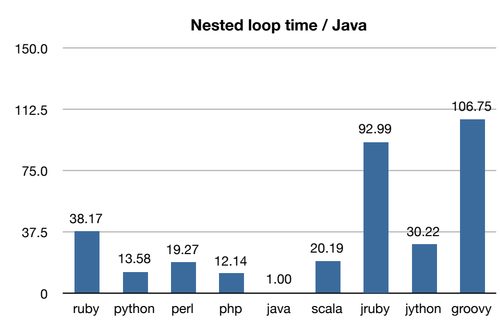

23

Boxing/unboxing

- Nested loops benchmark:
  - 12-107x slower than Java
- JRuby example:

```
for i in 1..n
    x = x + 1
end
```

x unboxed/reboxed at every iteration of loop



24

Iterators

Co-routine style iterators [CLU]

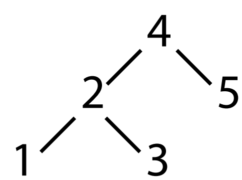
Each subsequent call to iterator (e.g., `elements`) resumes at previous `yield`

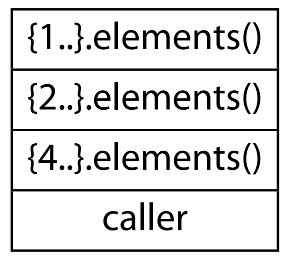
Efficient implementation of `yield` just adjusts stack pointer, but does not pop `elements` stack frame

On JVM: save iterator state on heap

```
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
```

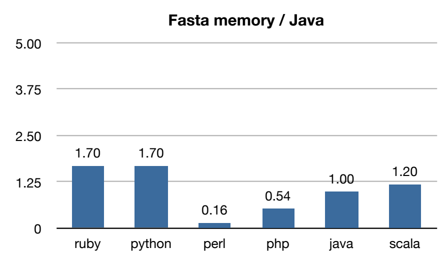


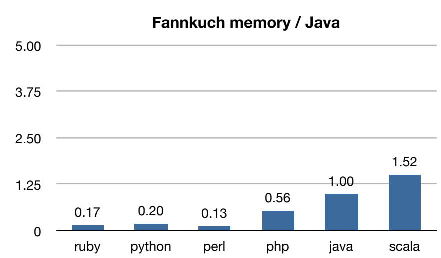


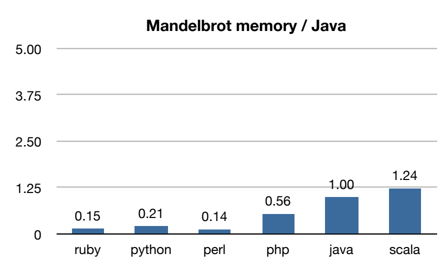
25

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







26

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

29

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?

30

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

31