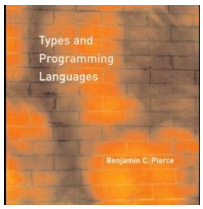
CS 565: Programming Languages

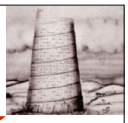
Lecture 1



Administrivia

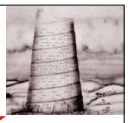
- Who am I?
- Course web page
- Office hours
- Main text
  - *Types and Programming Languages*, B. Pierce, MIT Press

Course Work



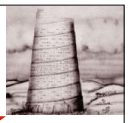
- Lectures
  - *optional, come if you are interested*
- Homeworks; Programming exercises; Midterm; Final
  - *none*
- Project
  - *work in groups <4*
  - *significant intellectual & programming challenge*
- Qualifying Exam
  - *5 questions covering theory of PL, based on the book*
  - *same level as previous years*
  - *sample questions available on request*

Prerequisites



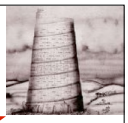
- Programming experience/maturity
  - *Knowledge of C*
  - *Undergraduate compilers and PL class (352 || 456 or equiv)*
- Mathematical maturity
  - *Familiarity w. first-order logic, set theory, graph theory, induction*
- Most important
  - *Intellectual curiosity and creativity*

Motivation



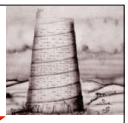
- Prove specific facts about programs
  - *Verify correctness*
  - *Safety or isolation properties*
- Understand specific language features
  - *Better language design*
  - *Guide improvements in implementations*

Goals



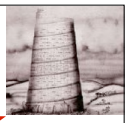
- A more sophisticated appreciation of programs, their structure, and the field as a whole
  - *Viewing programs as rich, formal, mathematical objects, not mere syntax*
  - *Define and prove rigorous claims about a program's meaning and behavior*
  - *Develop sound intuitions to better judge language properties*
- Develop tools to be better programmers, designers and computer scientists

Topics



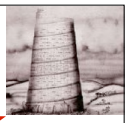
- Run-time systems, Virtual machines, interpreters, garbage collection, concurrency & multi-threading, synchronization
- Semantic formalisms, λ-calculus, introduction to types
- Simply-typed λ-calculus, records, references, subtyping, object-based programming
- Polymorphism, abstract data types, advanced topics (e.g., concurrency, linearity, ...)

Language Design



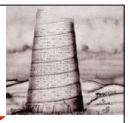
- Tower of Babel
  - Applications often have distinct (and conflicting) needs
  - AI (Lisp, Prolog, Scheme)
  - Scientific computing (Fortran)
  - Business (Cobol)
  - Systems programming (C)
  - Scripting (Perl, Javascript)
  - Distributed computation (Java)
  - Special-purpose (.....)
- Important to understand differences and similarities among different language features

Paradigms



- Imperative (Fortran, Algol, C, Pascal)
  - Designed around a notion of a program store
  - Behavior expressed in terms of transformations on the store
- Functional (Lisp, ML, Scheme, Haskell)
  - Programs described in terms of a collection of functions
  - “Pure” functional languages are state-free
- Logic (Prolog)
  - Programs described in terms of a collection of logical relations
- Concurrent (Fortran90, CSP, Linda)
- Special purpose (TeX, Postscript, HTML)

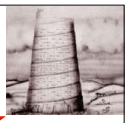
Project



- Implement a core JavaScript interpreter
- Design and implement language extensions
- The goal of this project is to give students hands-on experience with the challenges of designing and implementing real programming languages
  - *Work either in teams or individually*
  - *Pay attention to code quality and good practices*
  - *Best team wins a trip!*
- Warning:
  - *This is a challenging project*

Microsoft Research

Project

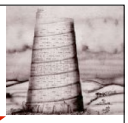


- Grading based on modules and team size
  - To get an A+ you need 100+(n-1)*20 points for a group of size n
- Modules

```
Core requirements:
 * [20] Parser  * [20] Interpreter  * [20] Runtime library
Add-ons
 * [20] Bytecode
 * GC:       - [20] mark-and-sweep or 2-space copying
             - [30] Generational or "clever"
 * Classes: - [20] alone
             - [30] Gradual typing
 * Threads  - [15] Complete wrapper around pthreads
             - [25] More clever native-feeling threads
             - [15] Thin locks
 * [30] JIT                * [25] Typed prototypes
 * [20] AJAX/XHR           * [25] TCP/IP
 * [15] C FFI              * [20] "Security"
 * [20] "Fast"             * [10] "Memory efficnt" interpreter
 * [10] Inline caches      * [20] Transactions
```

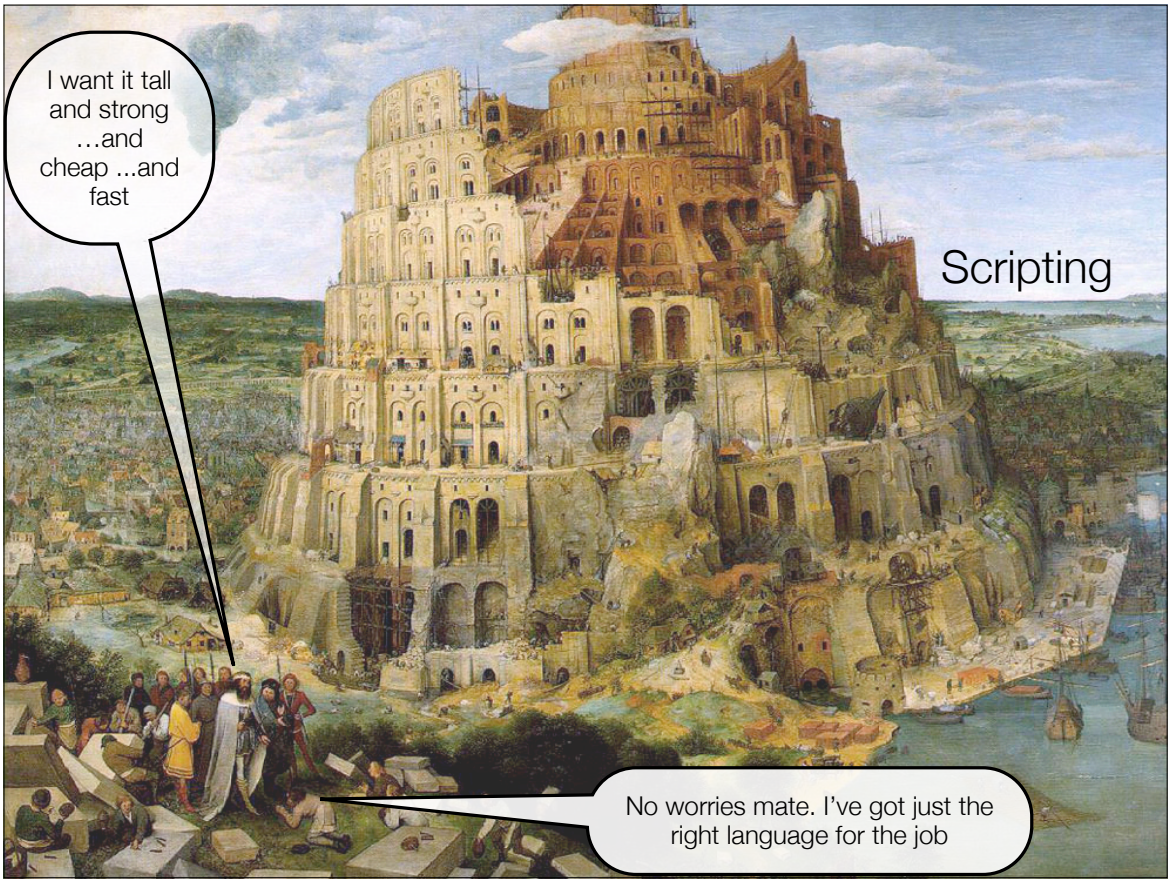
Microsoft Research

Project



- You will have to use a source version control system
- You will have to know how to program in C
- You will have to understand the basics of make
- You will use best practices for development
- You will write test cases
- The resulting software will be tested on a department machine
- The resulting software will run real JavaScript programs
- The resulting software will outperform low-end VMs

Microsoft Research



At the beginning...

- Interest in scripting language motivated by Pluto
  (aka in Swedish Premiepensionmyndigheten)
- Scripting languages are lightweight programming languages well suited for rapid development
- What sets them apart is both syntactic support for some domain and a very dynamic nature
- But are they appropriate for "real" projects?

...there was a script

- A modest Perl program hacked together to perform a simple data migration step while the grown ups built the real system
- Unfortunately, the real system was late, over budget, and unusable
- … and the script became the real system

...that grew up to be a program that...

- … manages the retirement savings of 5.5 million users
- … for a value of 23 billion Euros
- … with a team of 30 developers over 7 years

Pluto

320 000 lines of Perl

68 000 lines of SQL

27 000 lines of shell

26 000 lines of HTML

230 database tables

750 G bytes of data

24/7 availability

0 bugs allowed

The Road to Glory

- A number of factors contributed to the success of Pluto
    - High productivity of scripting languages
      Perl won over Java in all internal evaluations
    - Disciplined use of the language
      Many features disallowed by standards. Only C-like code.
      No floating points. No threads. No OO.
    - Fail fast, Abort, Undo
      Batch daily runs, undo all changes if an error is detected.
    - Contracts
      Home-brewed contract notation for Perl, runtime checked

Contracts for Perl

```
contract('do_sell_current_holdings')
  -> in(&is_person, &is_date)
  -> out(&is_state)
  -> enable;

sub do_sell_current_holdings {
   my ($person, $date)
   …
   if ($operation eq "BUD_") {
      …
   …
   return $state;
}
```

Lessons Learned

- The Pluto developers complained about
    - Syntax (it's ugly)
    - Typing (it's weak)
    - Speed (it's slow)
- Support for concurrency and parallelism is lacking
- Lack of encapsulation and modularity

The Questions are thus...

- Can we write dynamic scripts and robust programs in the same language?
- Can we go from scripts to programs and from programs to scripts freely?
- Can we do this without losing either the flexibility of scripting or the benefits that come with static guarantees?
- What kinds of static assertions about dynamic program behavior are most important to correctness?

Related Work

- *Lundborg, Lemonnier.* PPM or how a system written in Perl can juggle with billions. Freenix 2006
- *Lemonnier.* Testing Large Software With Perl. Nordic Perl Workshop 2007
- *Stephenson.* Perl Runs Sweden's Pension System. O'Reilly On Lamp, 2005

Thanks: Tobias Wrigstad

Dynamics in JavaScript

Understanding the dynamics of dynamic languages

- How dynamic should we, *must we*, be?
  Many anecdotal stories about need and use of dynamic features, but few case studies
  Are dynamic features used to make up for missing static features?
  Or are the programmers just "programmers"?
- Can we add a static type system to an existing dynamic language?
  without having to rewrite all legacy programs and libraries

Methodology

- We selected a very dynamic language, `JavaScript`, a cross between `Scheme` and `Self` without their elegance but with a large user base
- We instrumented a popular browser (`Safari`) and collected traces from the 100 most popular websites (`Alexa`) plus many other traces
- We ran an offline analysis of the traces to gather data
- We analyzed the source code to get static metrics

JavaScript is a Programming Language

- A familiar syntax

```
function List(v,n) {this.value=v; this.next=n;}

List.prototype.map = function(f){
  return new List( f(this.value),
                   this.next ? this.next.map(f) : null); }

var ls = new List(1, new List(2, new List(3, null)));

var nl = ls.map( function(x){return x*2;} );
```

JavaScript is a Programming Language

- A familiar syntax

```
function List(v,n) {this.value=v; this.next=n;}

List.prototype.map = function(f){
  return new List( f(this.value),
                   this.next ? this.next.map(f) : null); }

var ls = new List(1, new List(2, new List(3, null)));

var nl = ls.map( function(x){return x*2;} );
```

Constructor

field addition

JavaScript is a Programming Language

- A famili

adding a shared method

```
function List(v,n) {this.value=v; this.next=n;}

List.prototype.ma  = function(f){
  return new List(f(this.value),
                  this.next ? this.next.map(f) : null); }

var ls = new List(1, new List(2, new List(3, null)));

var nl = ls.map( function(x){return x*2;} );
```

JavaScript is a Programming Language

- A familiar syntax

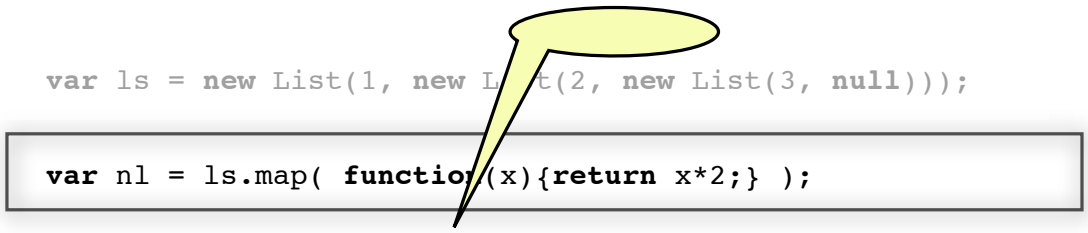
object

```
function List(v,n) {this.value=v; this.next=n;}

List.prototype.map = function(f){
  return new List(f(this.value),
                  this.next ? this.next.map(f) : null); }

var ls = new List(1, new List(2, new List(3, null)));

var nl = ls.map( function(x){return x*2;} );
```

JavaScript is a Programming Language

- A familiar syntax

```
function List(v,n) {this.value=v; this.next=n;}

List.prototype.map = function(f){
  return new List( f(this.value),
                   this.next ? this.next.map(f) : null); }

var ls = new List(1, new List(2, new List(3, null)));

var nl = ls.map( function(x){return x*2;} );
```



Challenges to static typing

- Lack of type declarations
- Eval and dynamic loading
- Addition/deletion of fields/methods
- Changes in the prototype hierarchy

Corpus

- 100 JavaScript programs were recorded

  Focus on the following:

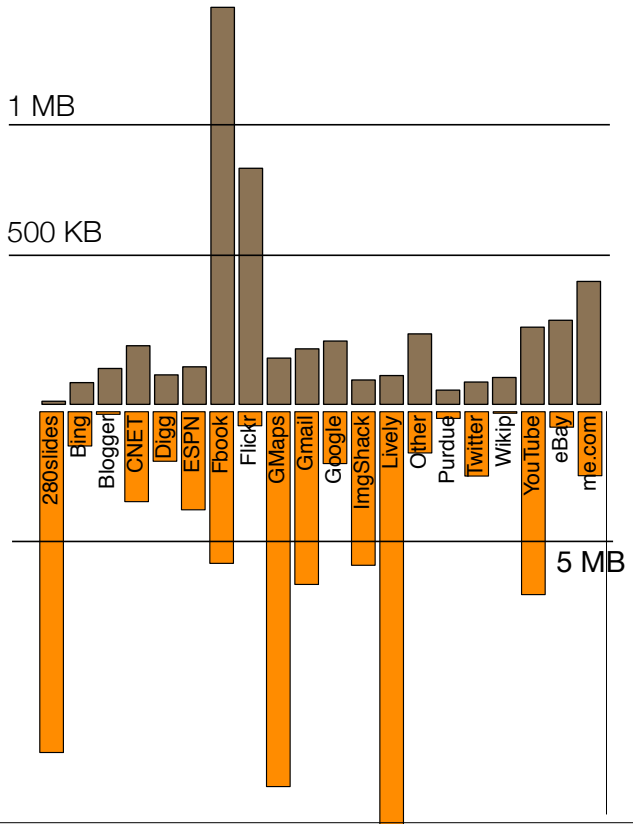
  280slides, Bing, Blogger, CNET, Digg, ESPN, Facebook, Flicker, GMaps, Gmail, Google, ImageShack, LivelyKernel, Other, Purdue, Twitter, Wikipedia, WordPress, YouTube, eBay, AppleMe

- The total size of the traces is 6.7 GB

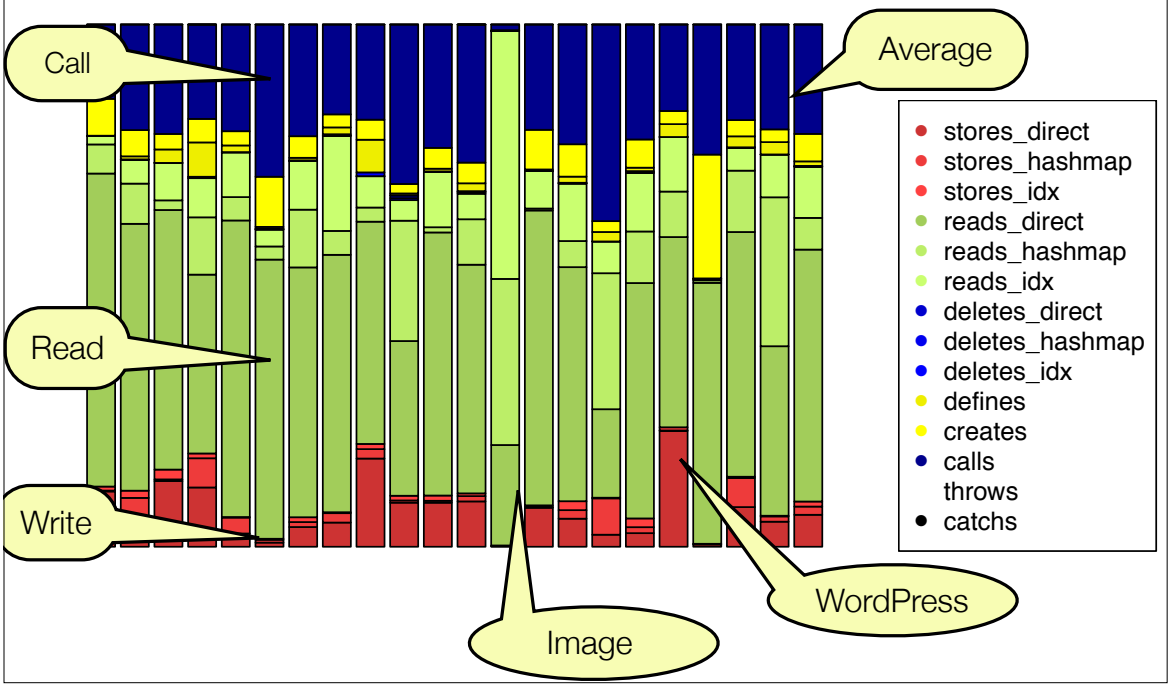
Corpus

- Size of source in bytes

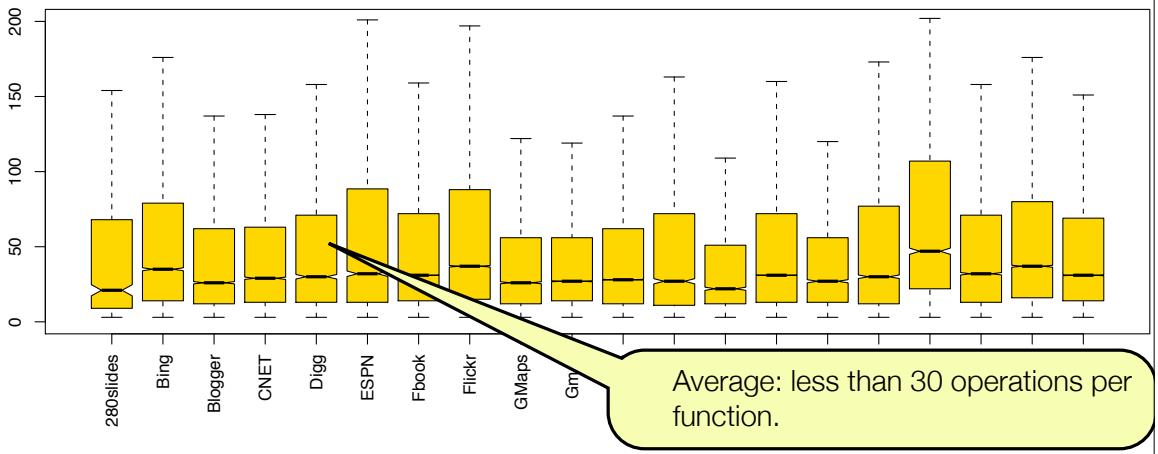
- Size of average trace in bytes



Instruction Mix



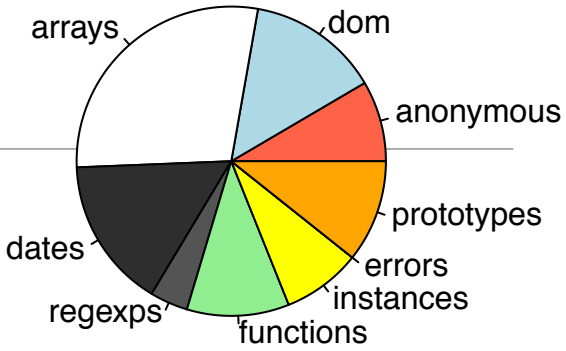
Function Size

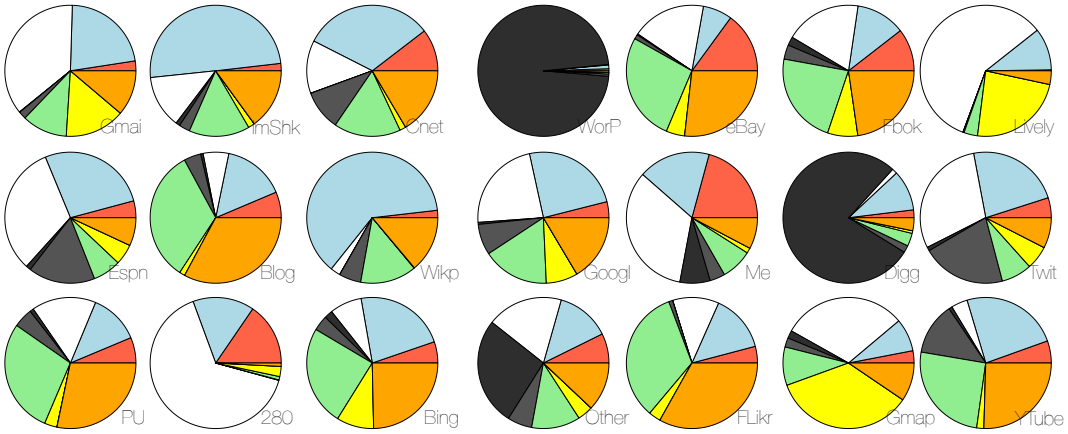


- Number of instruction executed within each function

Live Data

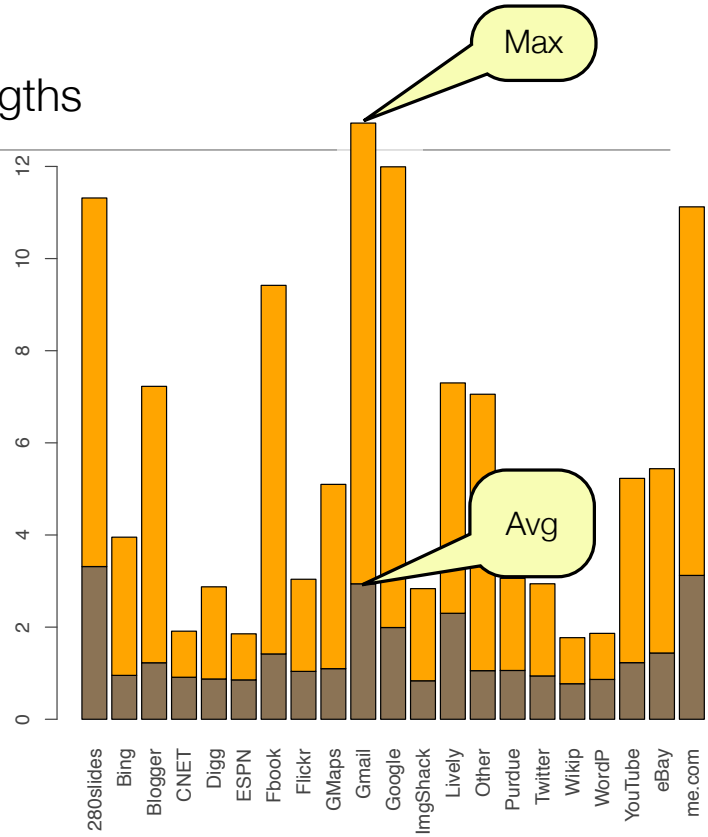
- Objects allocated in the traces broken down in major categories





Prototype chain lengths

- Prototype chains allow sharing behavior in a more flexible way than inheritance
- Prototype chain length similar to inheritance depth metrics
- While, average close to 1, maximum depends on coding style

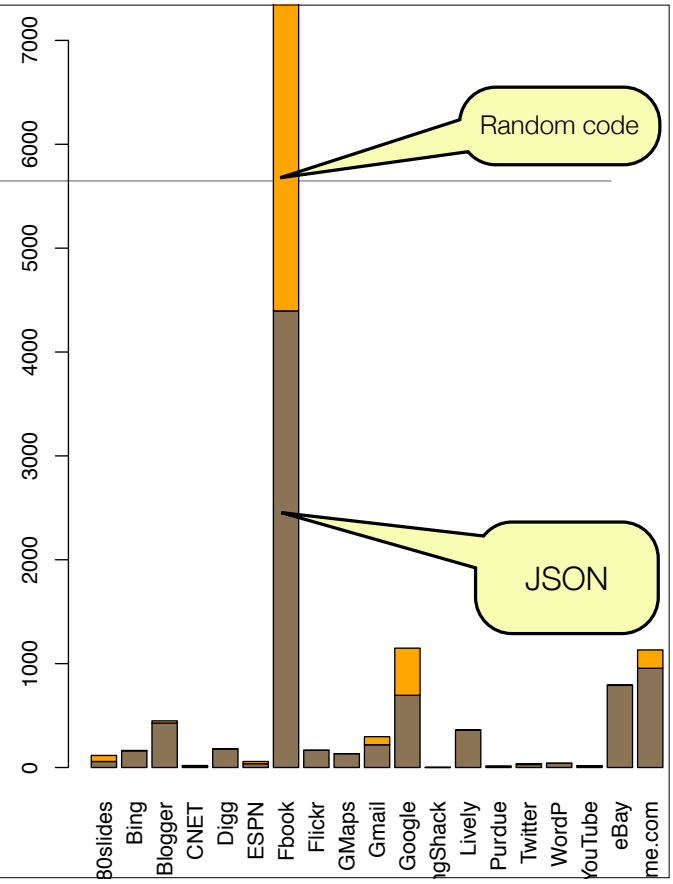



```
x._proto_ = y
```

Oh Eval, my Eval

- Eval can perform arbitrary damage to data structures
- What if most evals were deserialization of JSON data?

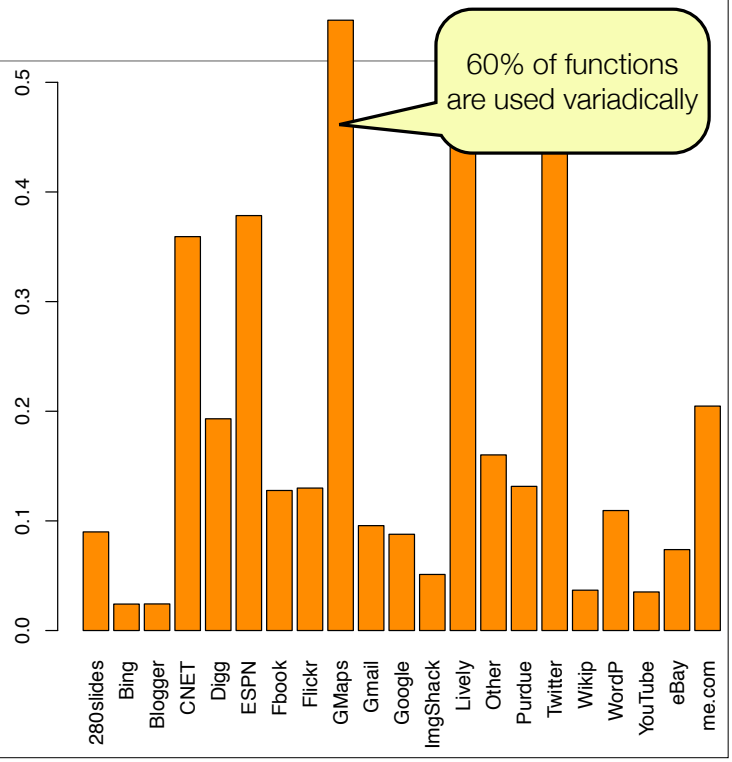
```
eval("x.f = 3")
```



Variadicity

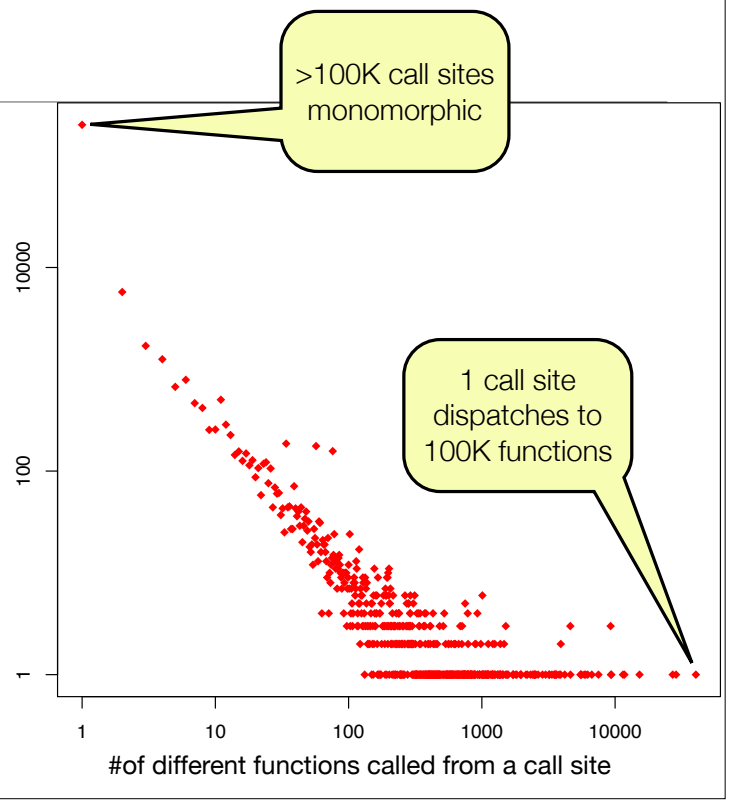
- Functions need not be called with the "right" number of arguments
- Missing args have value `UNDEFINED`, additional args accessed by position

```
f(1)        # too few
f(1,2,3)    # too many

f = function(x,y) {…}
```



Dynamic dispatch

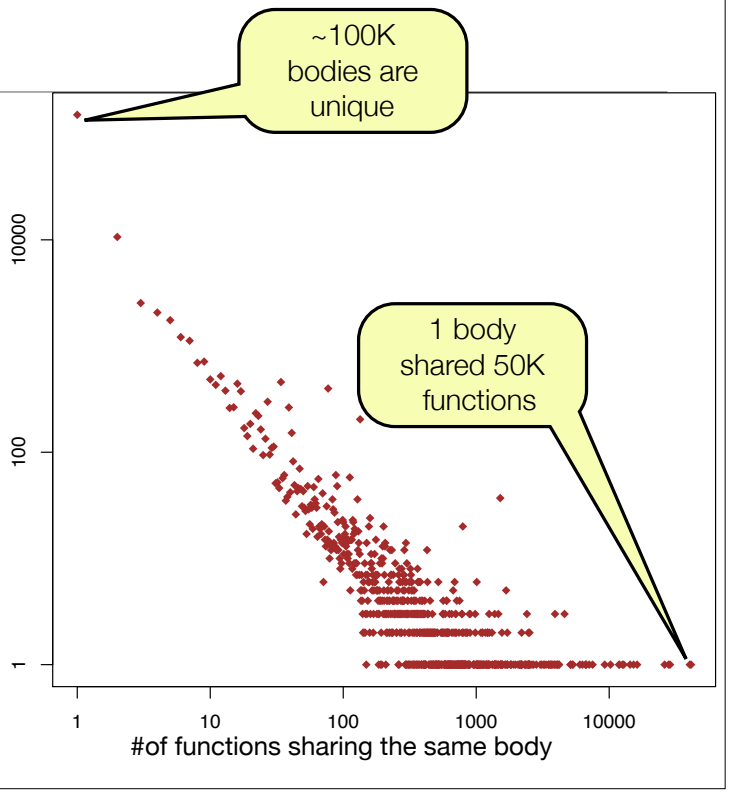
- Dynamic binding is a hallmark of object-oriented languages
- How dynamic is our corpus?



Dynamic dispatch

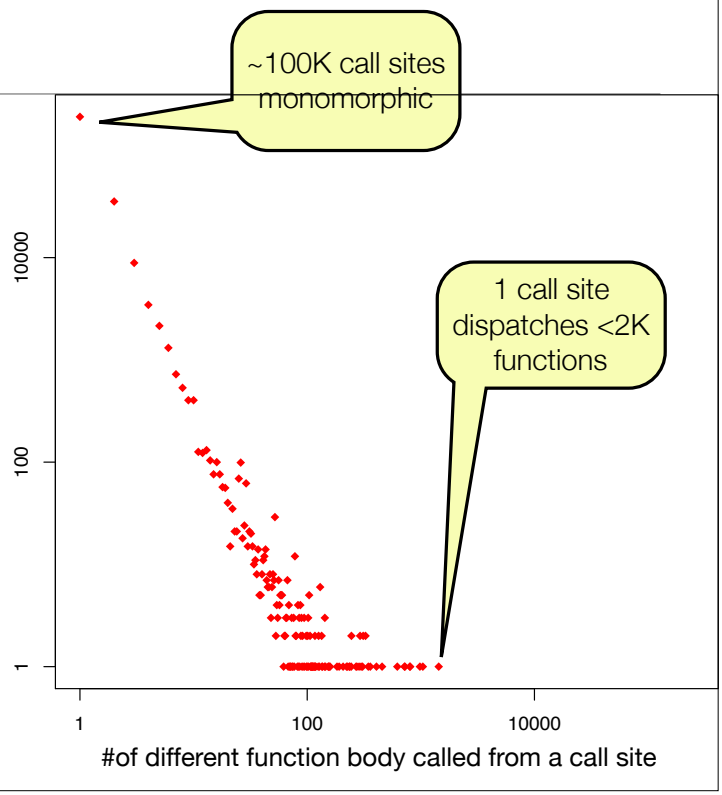
- What if there were many identical functions?
- Programming style (or lack thereof) matters

```
function List(h,t){
 this.head=h;
 this.tail=t;
 this.c=function(l){…}
}
```



Dynamic dispatch

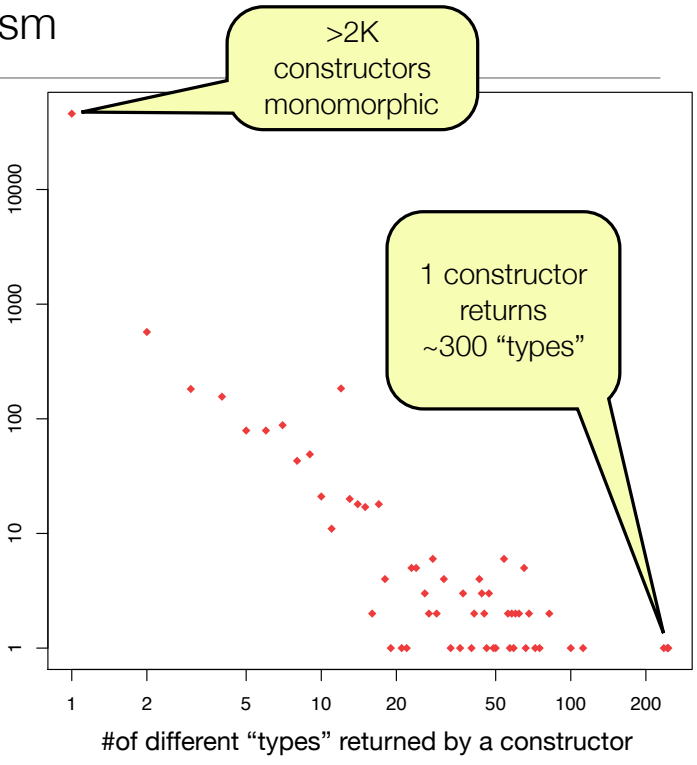
- These are not so different from Java or C#…



Constructor dynamism

- Constructors are “just” functions that side-effect `this`.
- Accordingly a constructor can return different “types”, i.e. objects with different properties

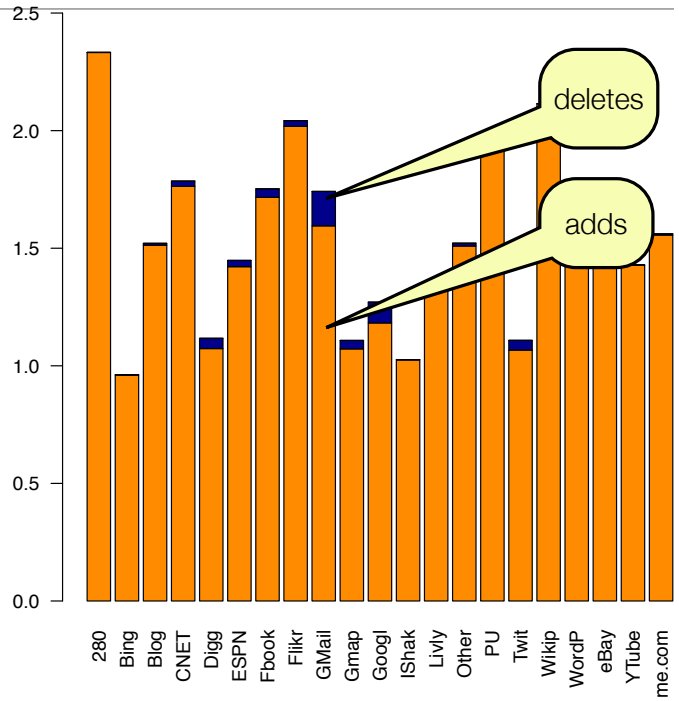
```
function Person(n,M){
  this.name=n;
  this.sex=M;
  if(M){
   this.likes= “guns”
  }
}
```



Addition/Deletion

- JavaScript allows runtime addition and deletion of fields and methods in objects (including prototypes)
- Ignoring constructors, the average number of additions per object is considerable
- Deletion are less frequent but can't be ignored

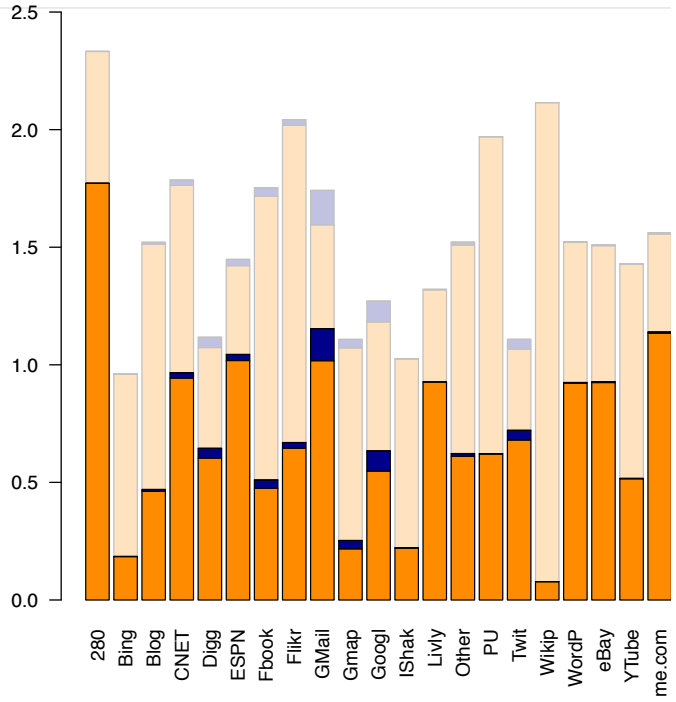
```
x._proto_.f = F

delete x.y
```



Addition/Deletion

- Programming style (or lack thereof) matters.
- What about objects constructed by extending an empty object?
- Heuristic: construction ends at first read

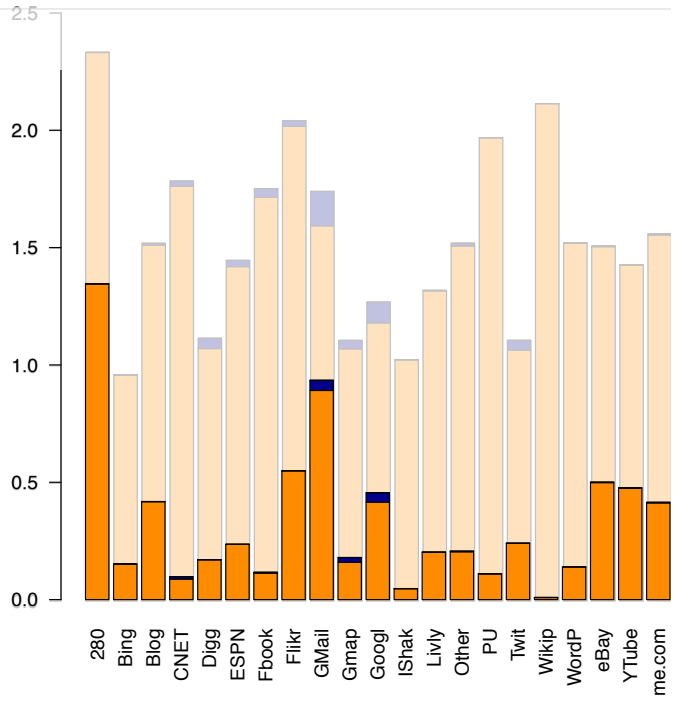
```
x = {}

x.head="Mickey"
x.map=function(x){…}

… = x.f …
```



Addition/Deletion

- Hash tables & arrays are objects, and *vice versa*
- What if most of add/ deletes were hash/index operation?
- Heuristic: syntax of access operations

```
… x[name] …

… x[3]…
```



Conclusions

- JavaScript programs are indeed dynamic
- All features are used, but not all the time
- Some of the abuse is sloppy programming
- *Hope of imposing types on legacy code?*

Related Work

- *Anderson, Giannini, Drossopoulou.* Towards type inference for JavaScript. ECOOP 2005
- *Mikkonen, Taivalsaari.* Using JavaScript as a Real Programming Language. TR SUN 2007
- *Holkner, Harland.* Evaluating the dynamic behavior of Python applications. ACSC 2009
- *Chugh, Meister, Jhala, Lerner.* Staged information flow for JavaScript. PLDI 2009
- *Furr, An, Foster, Hicks.* Static type inference for Ruby. SAC 2009
- *Guha, Krishnamurthi, Jim.* Using static analysis for Ajax intrusion detection. WWW 2009
- *Jensen, Møller, Thiemann.* Type analysis for JavaScript. SAS 2009

Current team: Sylvain Lebresne, Gregor Richards, Brian Burg
Alumni: Johan Ostlund, Tobias Wrigstad
Sponsor: ONR, NSF

Performance of Scripting Languages

Slides by Nate Nystrom

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

*or *will support* optional typing; we haven't implemented it yet

Dynamic language implementation

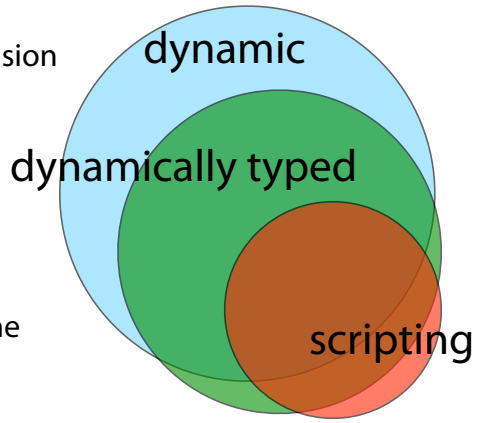
Decided early on to implement Thorn on the JVM

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”);
```

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
```

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

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

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

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

Thorn
```
val m = Map();
m("one") = 1;
```


Examples

Java
```
List<Integer> l = Arrays.asList(new int[] { 1,2,3 });
l.add(4);
```

Scala
```
val l = [1,2,3];
l += 4;
```

Ruby
```
l = [1,2,3]
l.push(4)
```

Python
```
l = [1,2,3]
l.append(4)
```

Perl
```
@l = (1,2,3);
push(@l, 4);
```

PHP
```
$m["one"] = 1;
```

Groovy
```
def l = [1,2,3]
l.add(4)
```

Thorn
```
var l = [1,2,3];
l += [4];

val l = List[1,2,3];
l.add(4);
```

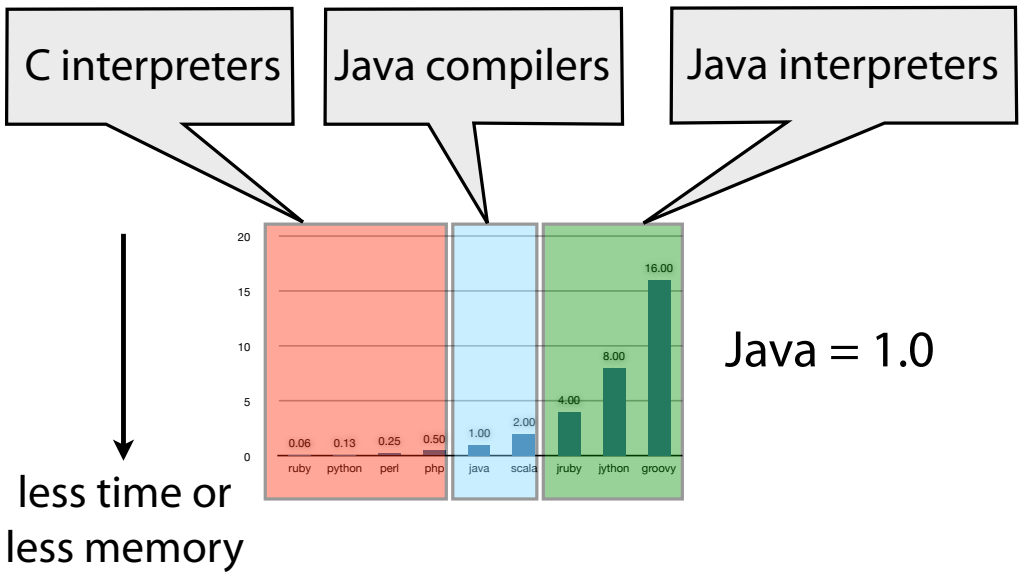
Performance

- 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

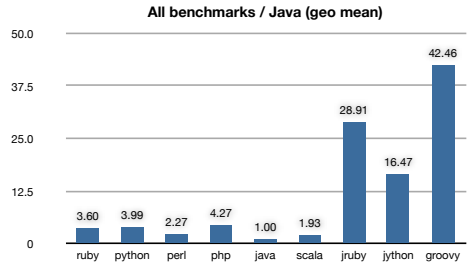
- Setup:
  - Macbook Pro, 2.4GHz Intel Core Duo, 2GB RAM
  - JVM: HotSpot JVM 1.5.0, 512MB heap

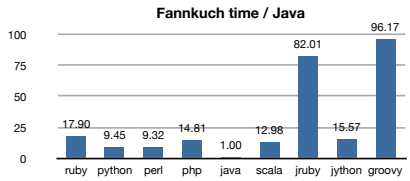
*http://shootout.alioth.debian.org

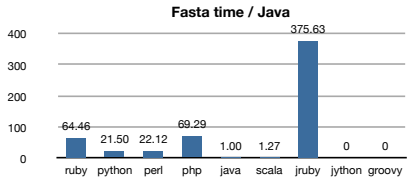
Reading the graphs

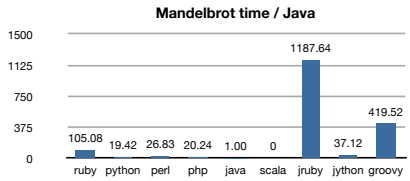


Run times / Java









- 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

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

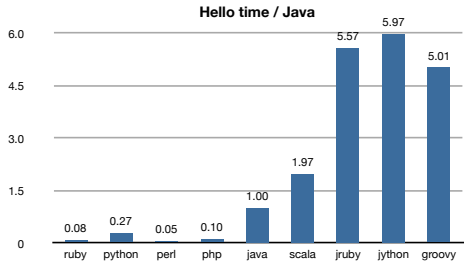
Object model mismatch

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
```

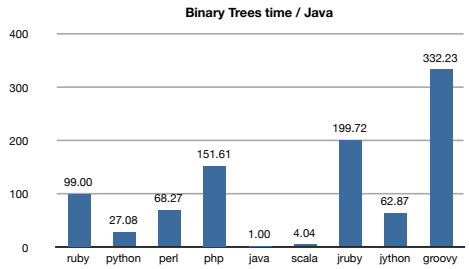
```
>>> x = MyClass()
>>> x.f
1
>>> x.get()
1
>>> x.g = 'a'
>>> x.g
'a'
```

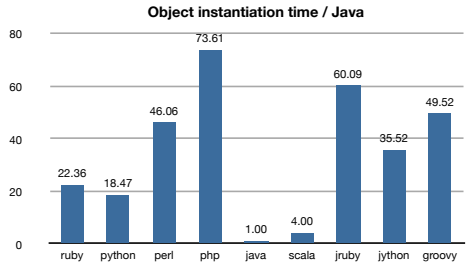
Objects implemented as hash tables

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

Objects

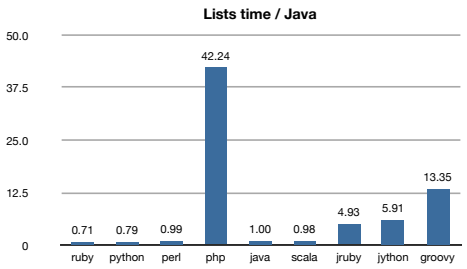
- Binary tree creation, traversal
  - C interpreters
    - 27-152x Java
  - Java interpreters
    - 63-332x Java
- Object instantiation
  - C interpreters
    - 18-74x Java
  - Java interpreters
    - 35-60x Java

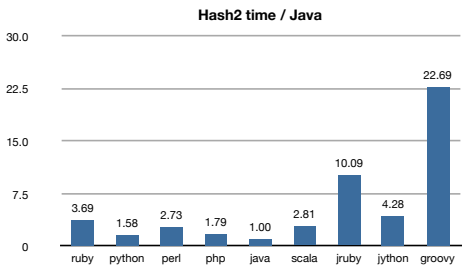




Data structures

- 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

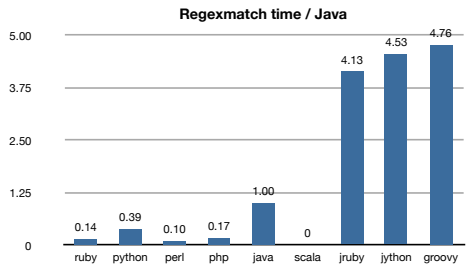
Must box primitive values

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

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

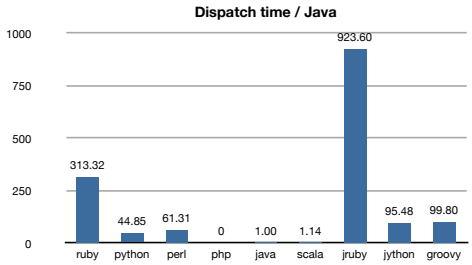

Strings

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



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

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

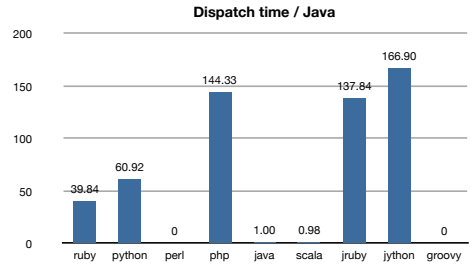
```
class C {
  def m() = 0;
}

x.m();
```

⇨

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

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

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

Dispatch time

- Dispatch benchmark
  - 40-167x Java
- JRuby, Jython:
  - dispatch through hash tables
- Groovy:
  - call methods via reflection API

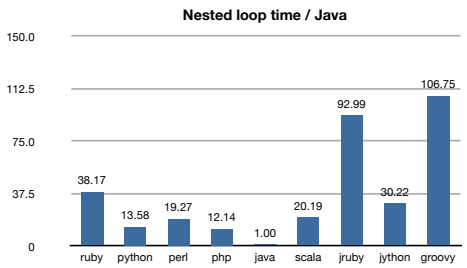


Boxing/unboxing

- 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



Iterators

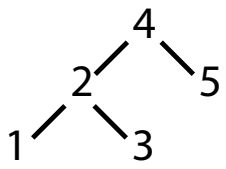
Co-routine style iterators [CLU]

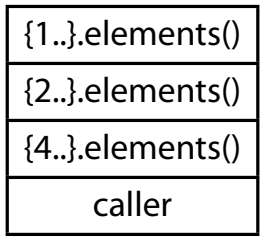
Each subsequent call to iterator (e.g., `elements`) resumes at previous `yield`

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

On JVM: save iterator state on heap

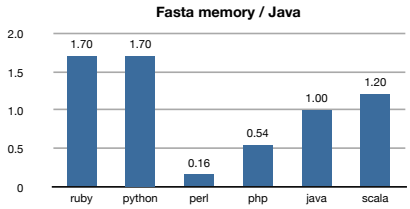
```
def elements(self):
    for x in self.left.elements():
        yield x
    yield self.value
    for x in self.right.elements():
        yield x

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

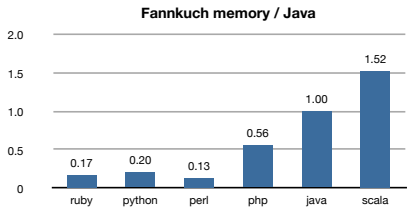


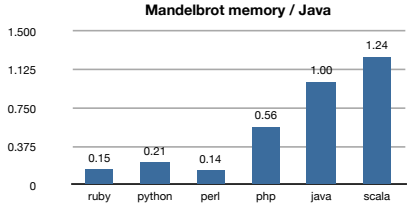


Peak memory usage

- 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

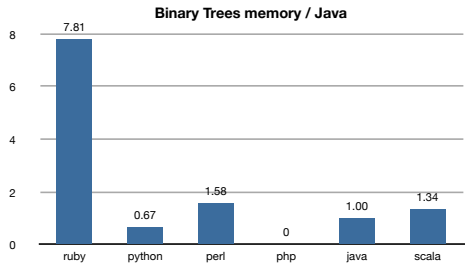


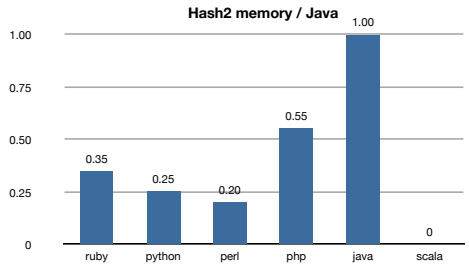




Memory

- Java objects often smaller
- Scripting language objects implemented as hash tables (multiple objects on JVM)
- Java hashes larger





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?

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

VIRTUAL EXECUTION ENVIRONMENTS

Jan Vitek

PURDUE
UNIVERSITY

with material from Nigel Horspool and Jim Smith

(S³)