Of Scripts and Programs

Tall tales, Urban Legends and Future Prospects

Jan Vitek

S3lab, Purdue University

based on joint work with

Bard Bloom John Field IBM

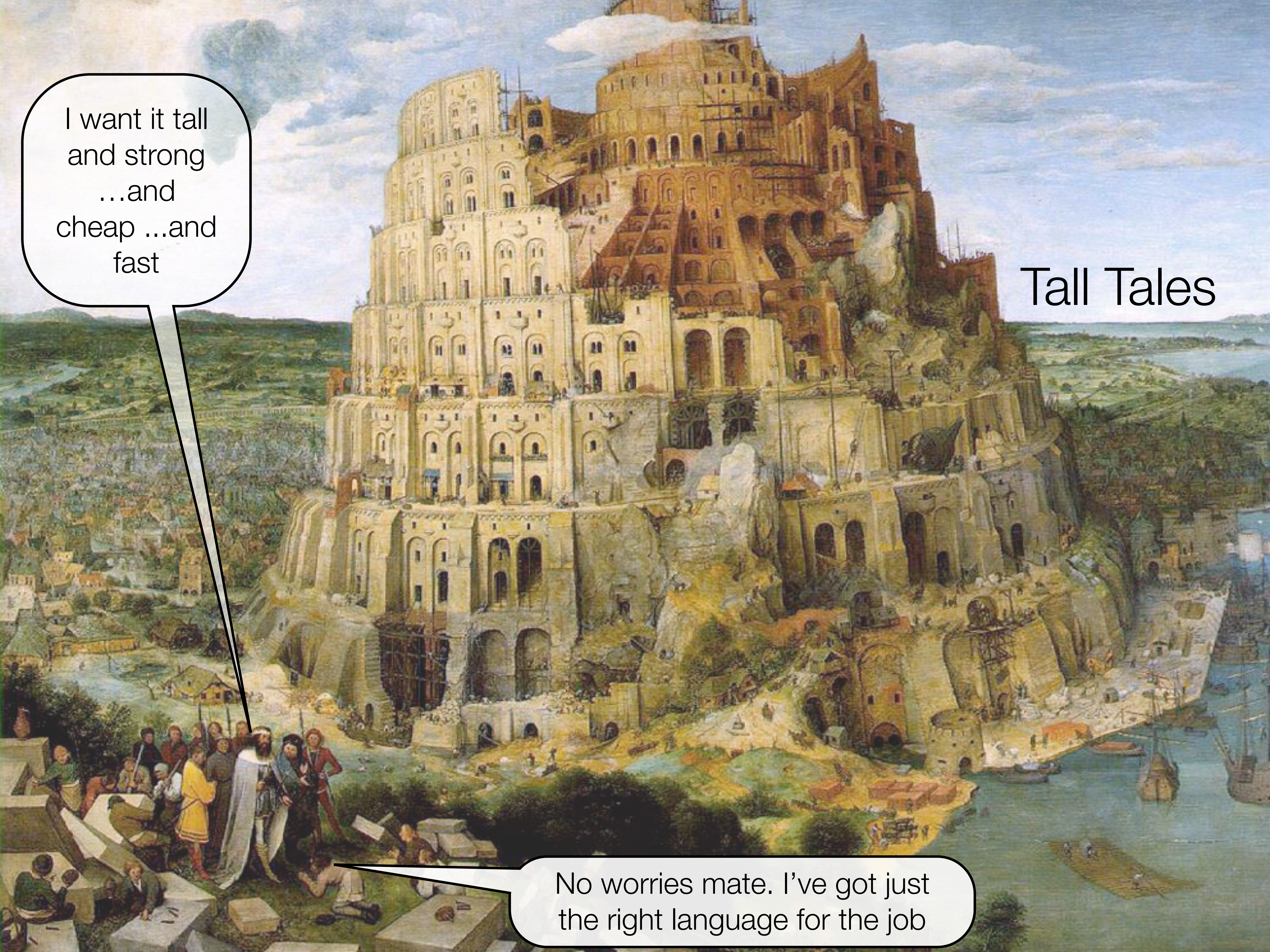
Nate Nystrom U. Texas Arlington

Francesco Zappa Nardelli INRIA

Tobias Wrigstad Stockholm University

Brian Burg Nicholas Kidd Sylvain Lebresne Johan Östlund Gregor Richards Purdue

Tall Tales


I want it tall and strong …and cheap ...and fast
No worries mate. I’ve got just the right language for the job

At the beginning...

- Our work motivated by Tobias' involvement with Pluto
  (aka in Swedish Premiepensionmyndigheten)
- We have been trying to get code for three years,
  but it's too sensitive to release...
- So, apply salt to the following slides,
  second hand material only, based on
  discussions with Tobias and slides
  by Lemonnier and Lundborg,
  errors are mine, all mine.

...there was a script

- A modest Perl program hacked together to perform a simple data migration step while the grown ups (BIC = Big IT Company) built the real system
- Unfortunately, the real system was late, over budget, and unusable
- So, BIC got fired 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?
- *Can I have my cake and eat it too?*

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

Urban Legends

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;}
```

Constructor

field addition

```
List.prototype.map = function(f){
  return           f(this.value),
                   this.next ? this.nex            ull); }

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 familia

adding a shared method

```
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;} );
```

higher order function

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                ue),
                   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;} );
```

object construction

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;} );
```

Closure

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:

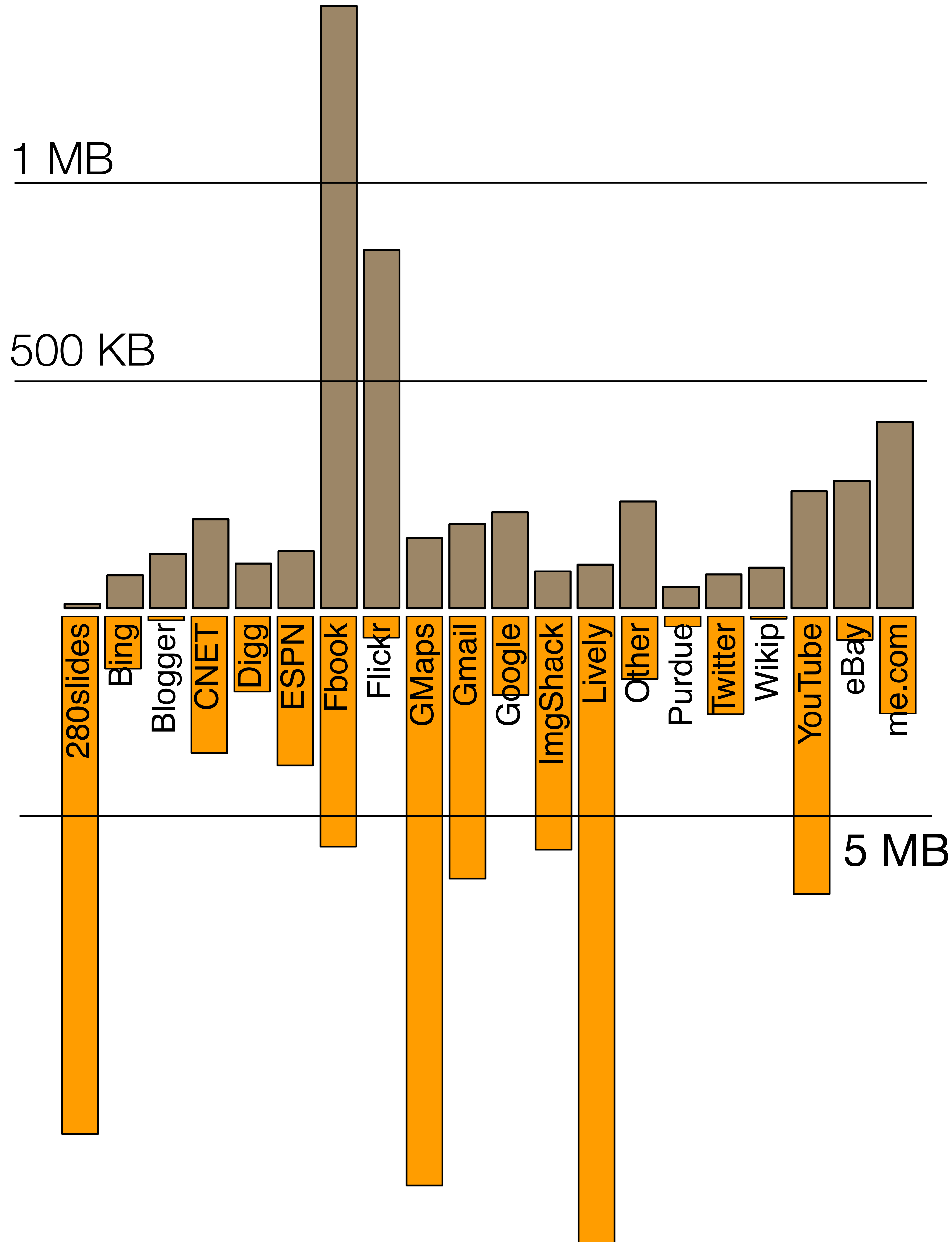
  280slides, Bing, Blogger, CNET, Digg, ESPN, Facebook, Flicker, GMaps, Gmail, Google, ImageShack, LivelyKernel, Other, Purdue, Twitter, Wikipedia, WordPress, YouTube, eBay, AppleMe

- The total size of the traces is 6.7 GB

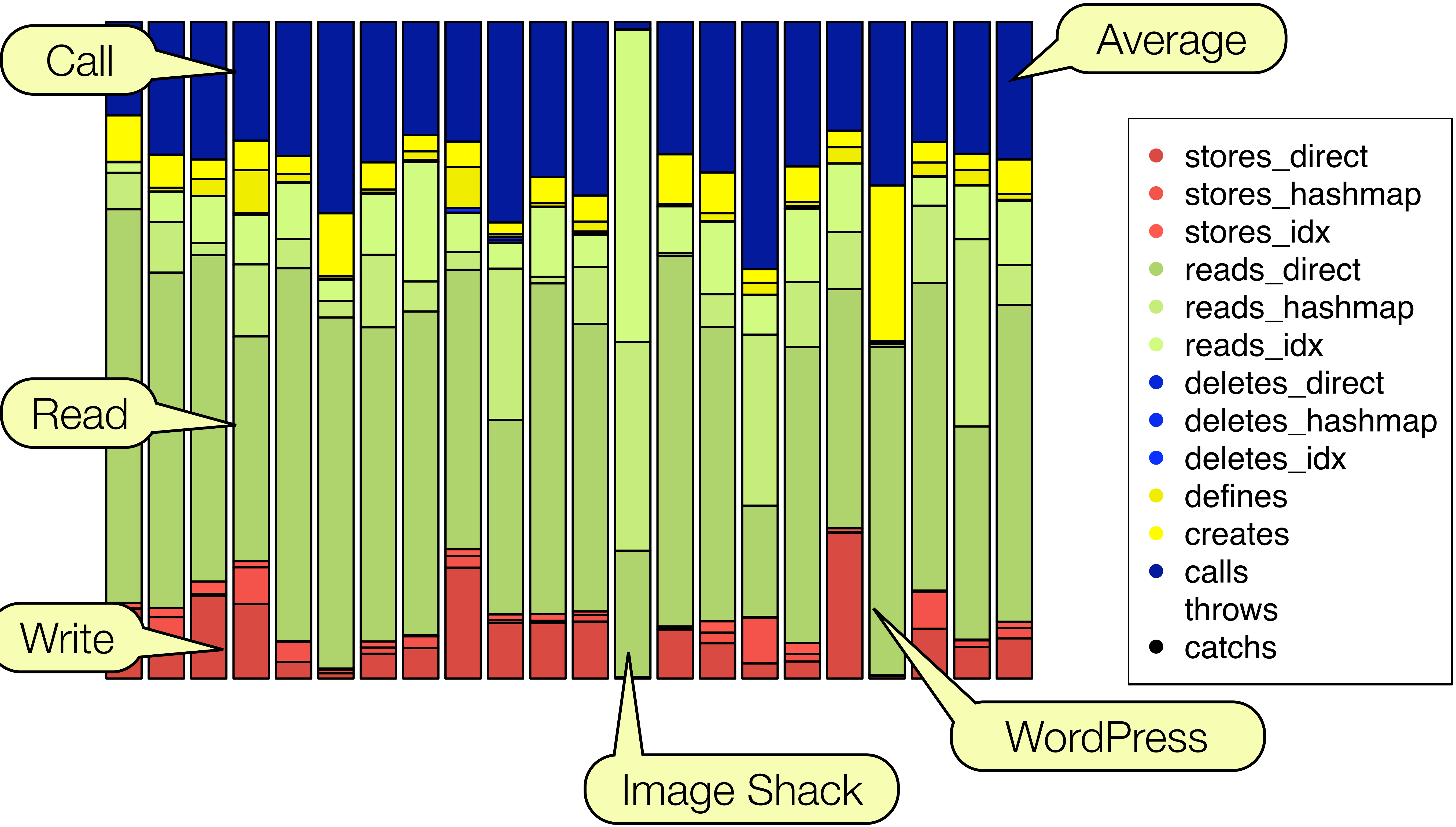
Corpus

- Size of source in bytes

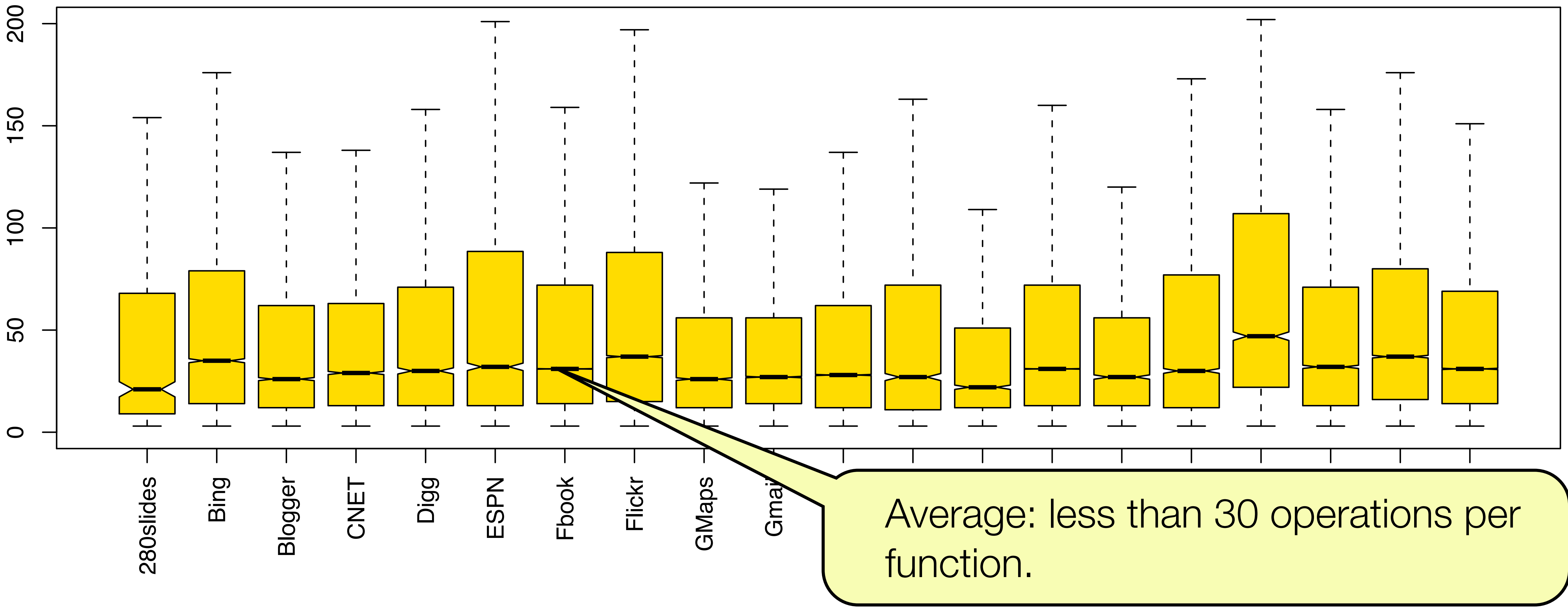
- Size of average trace in bytes



Instruction Mix



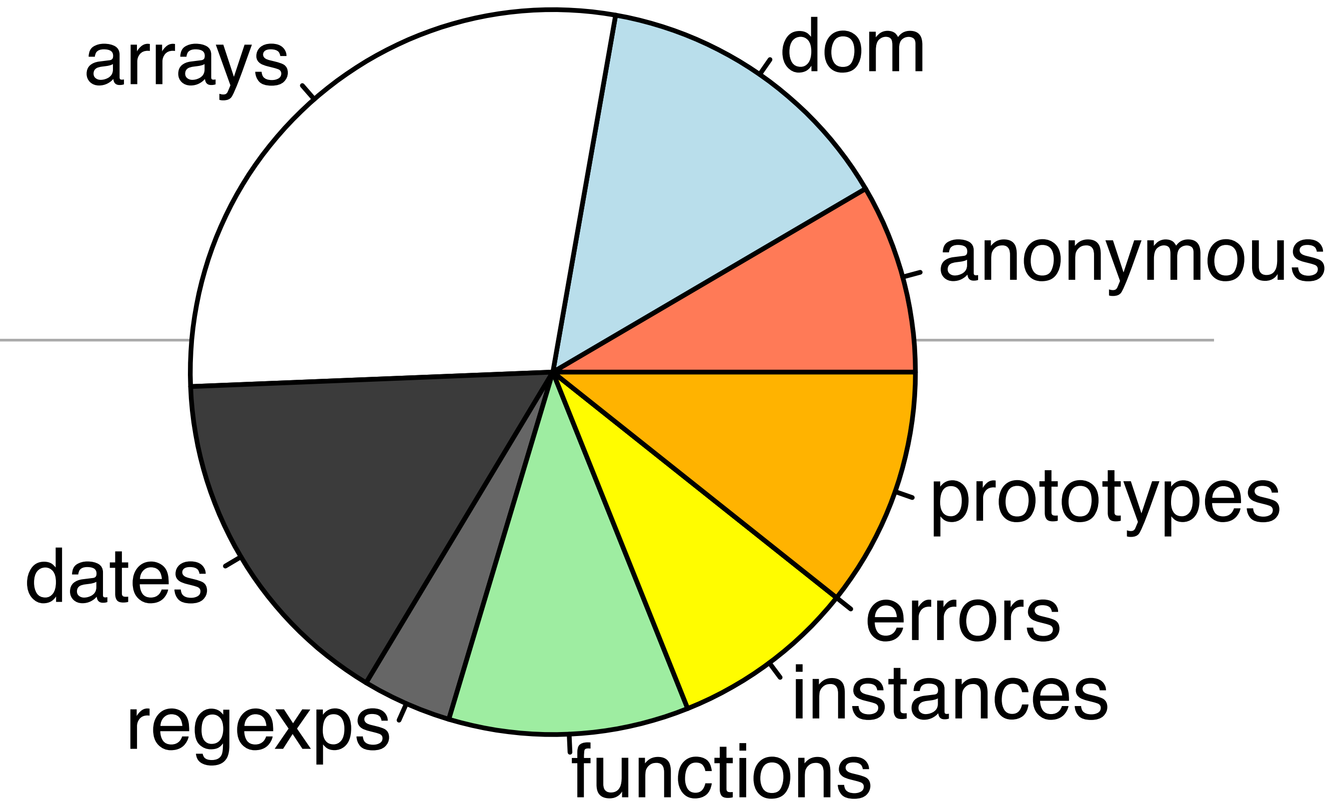
Function Size

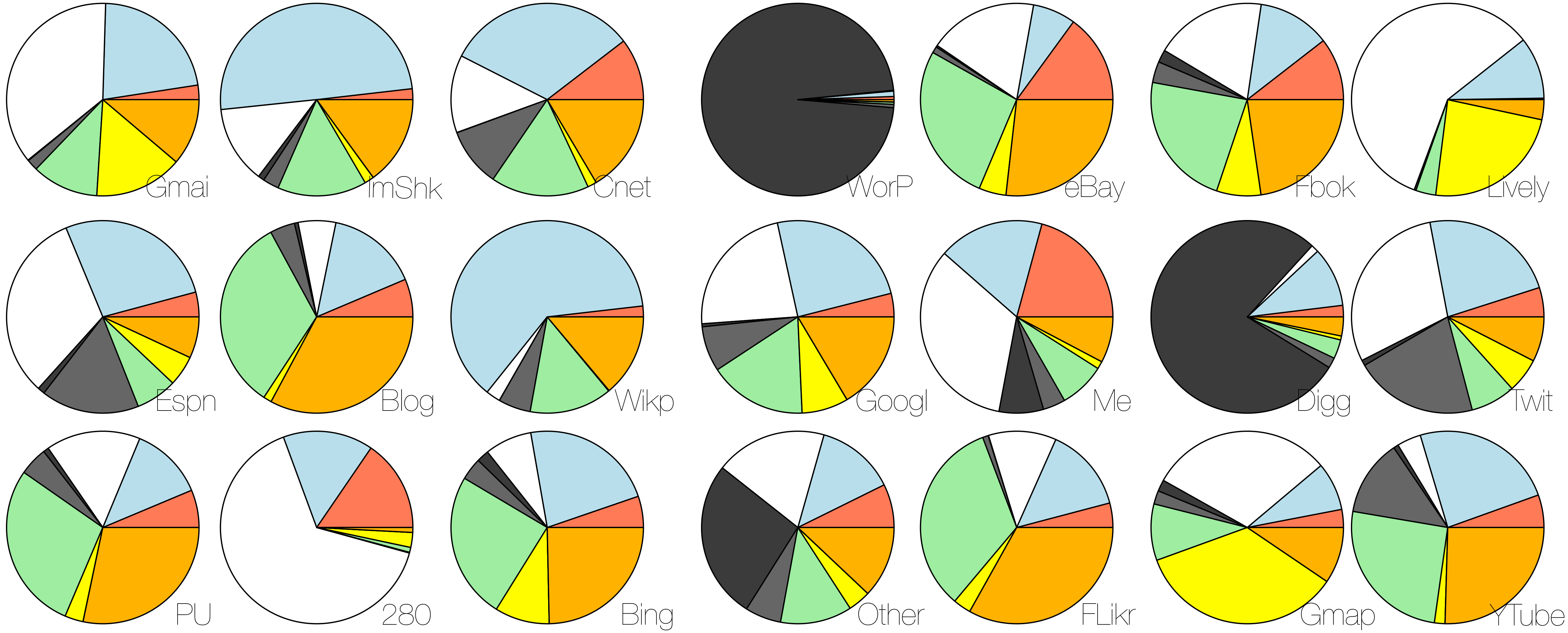


- Number of instruction executed within each function

Live Data

- Objects allocated in the traces broken down in major categories

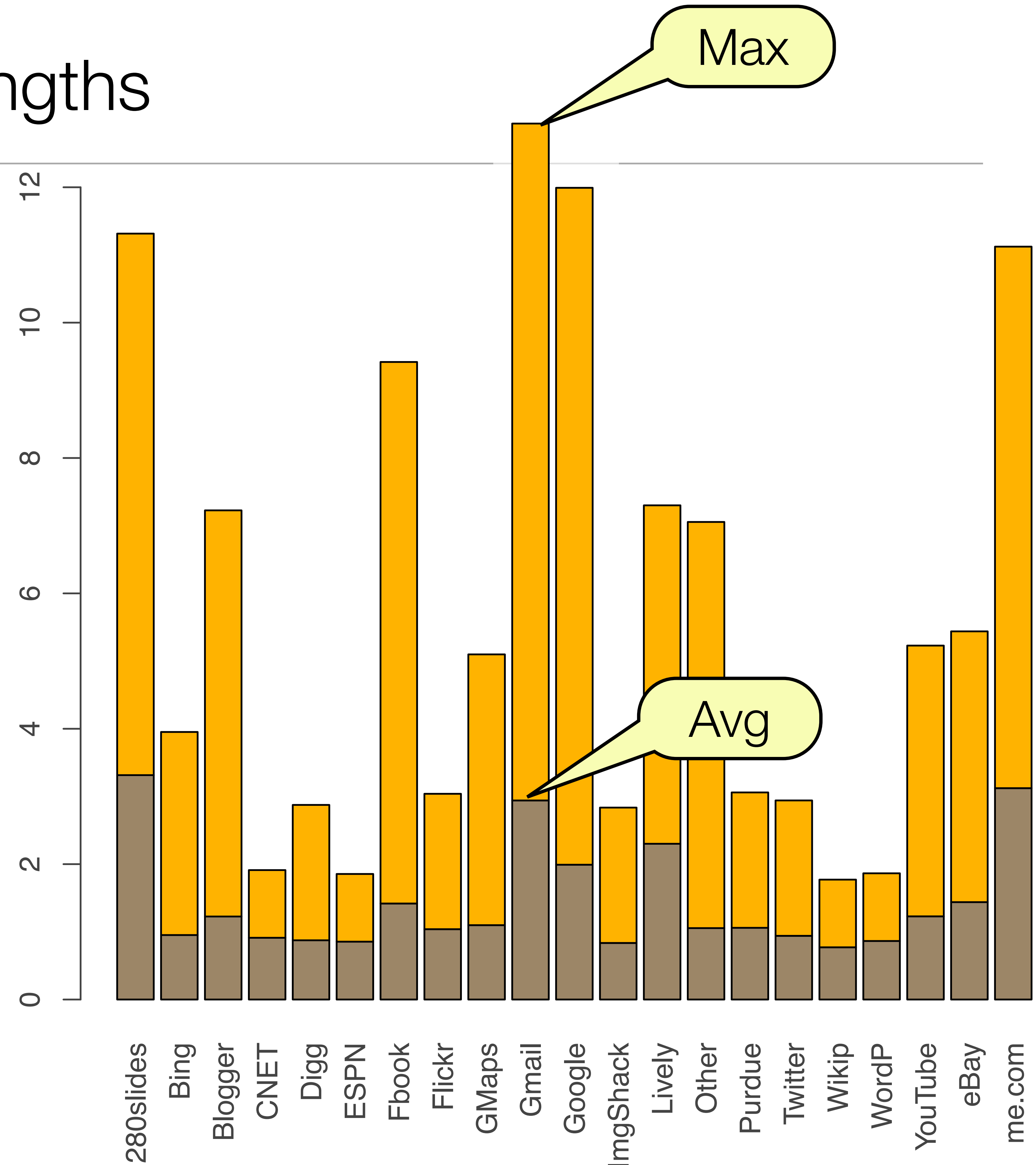




Prototype chain lengths

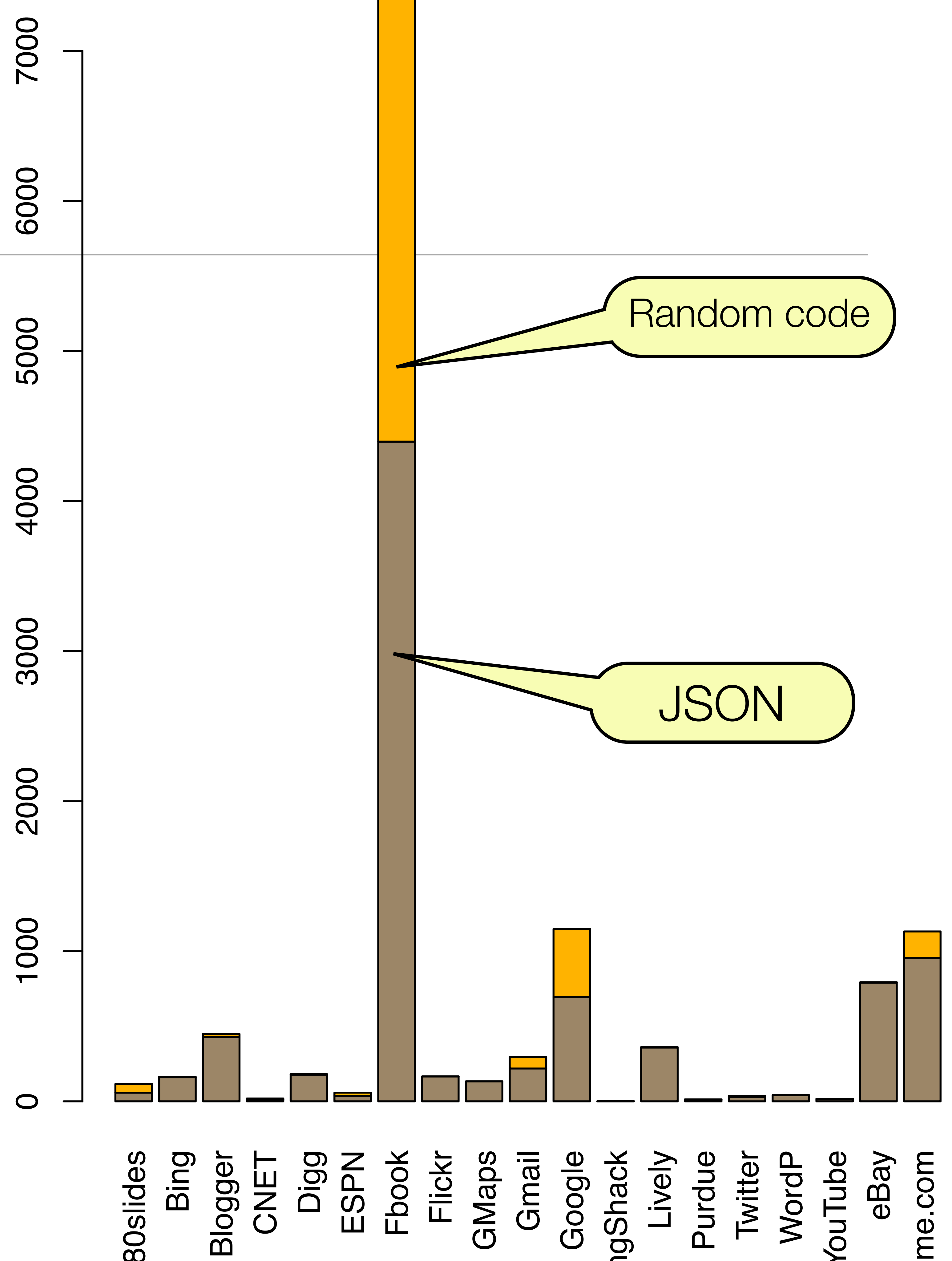
- Prototype chains allow sharing behavior in a more flexible way than inheritance
- Prototype chain length similar to inheritance depth metrics
- While, average close to 1, maximum depends on coding style

```
x._proto_ = y
```



Oh Eval, my Eval

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



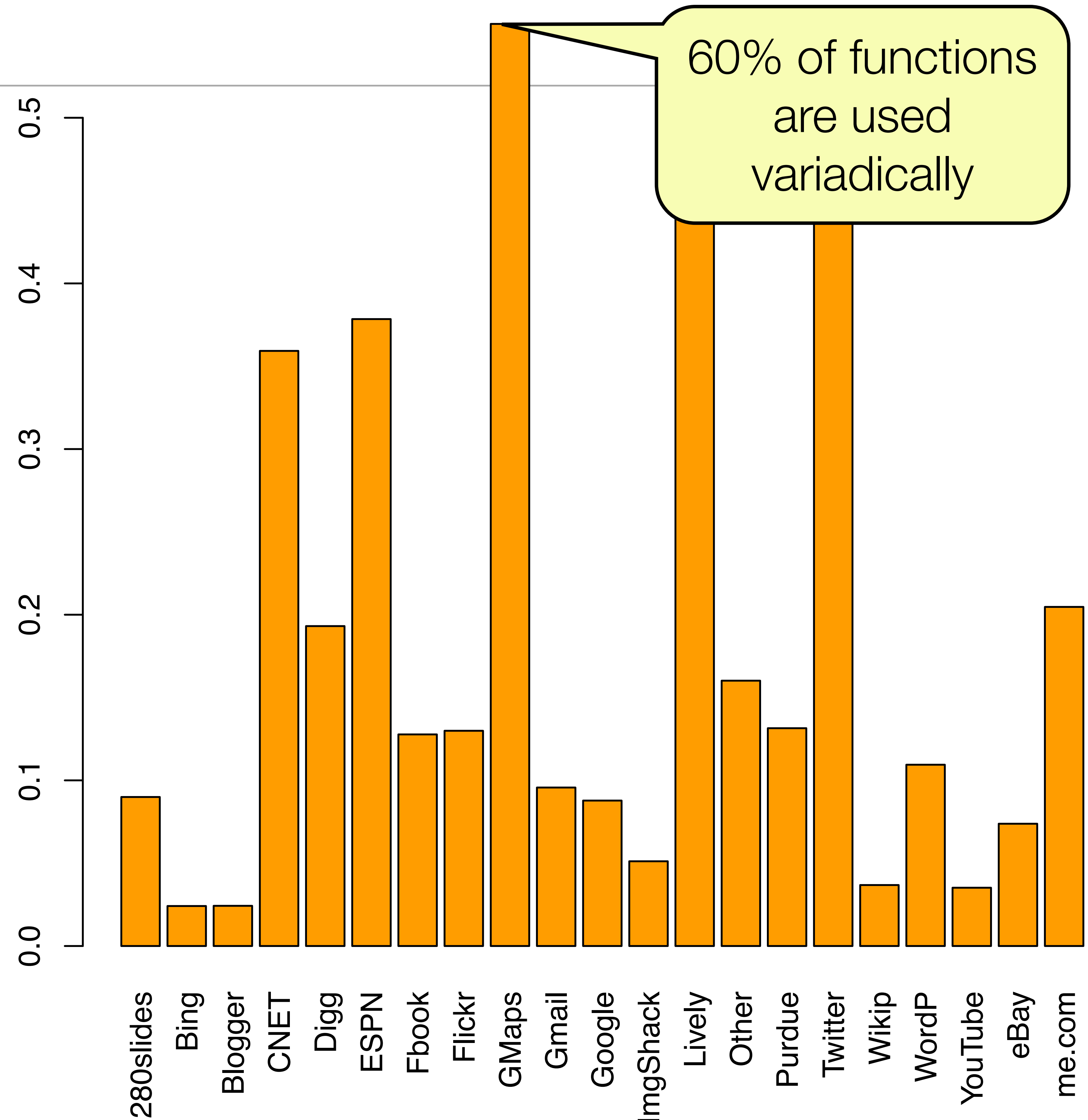

```
eval("x.f = 3")
```

Variadicity

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

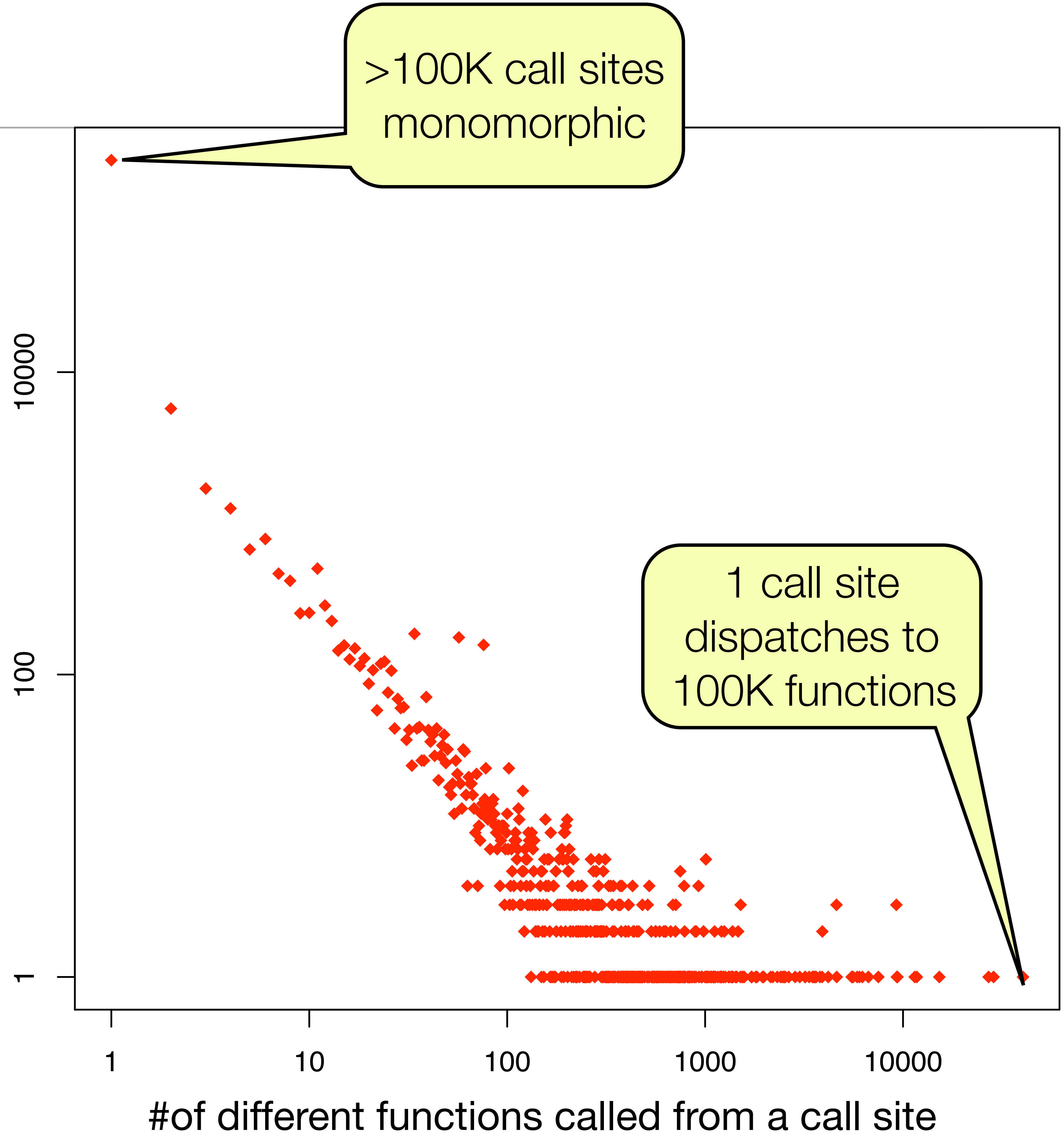
```
f(1)          # too few
f(1,2,3)      # too many

f = function(x,y) {…}
```



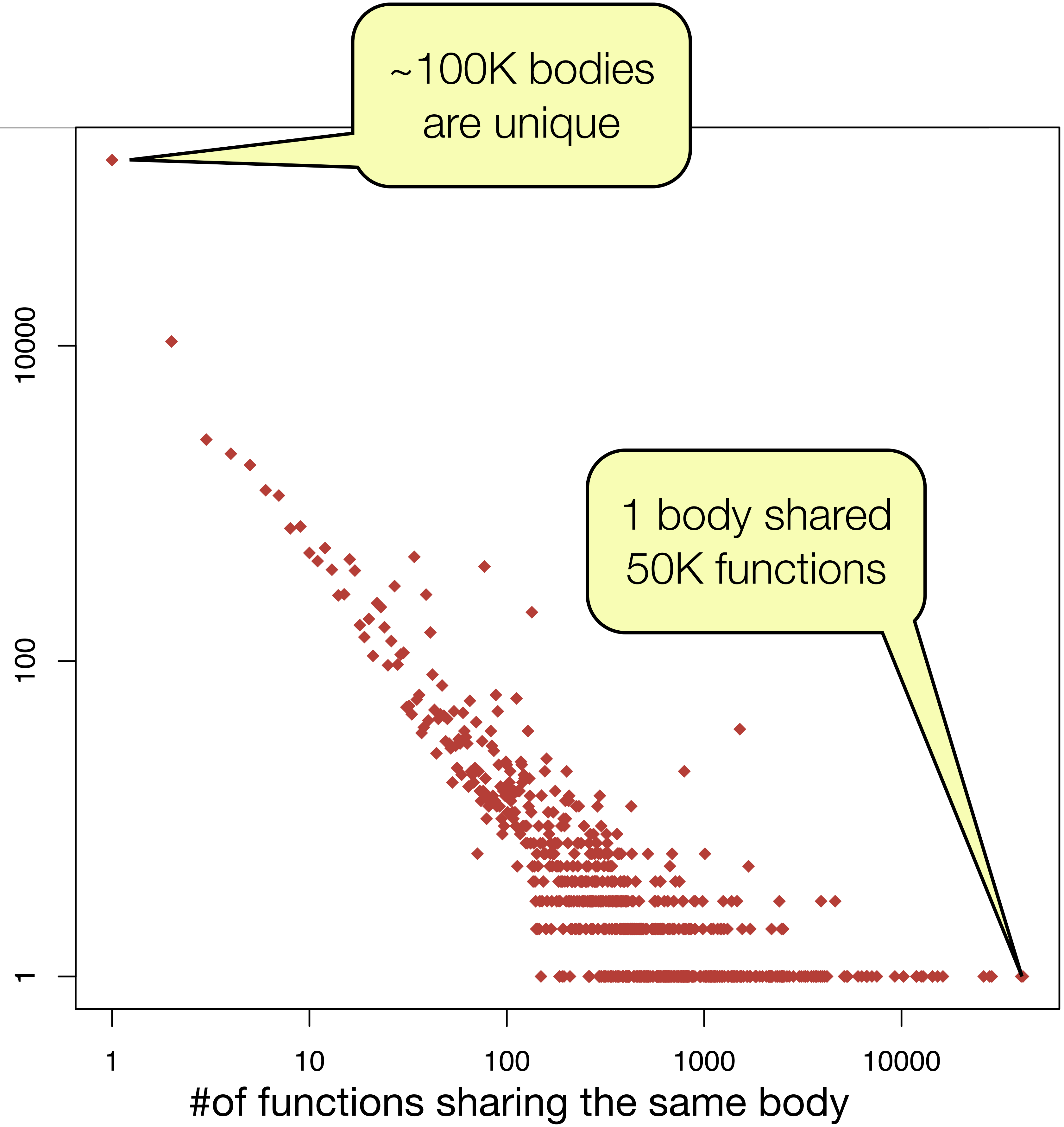
Dynamic dispatch

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



Dynamic dispatch

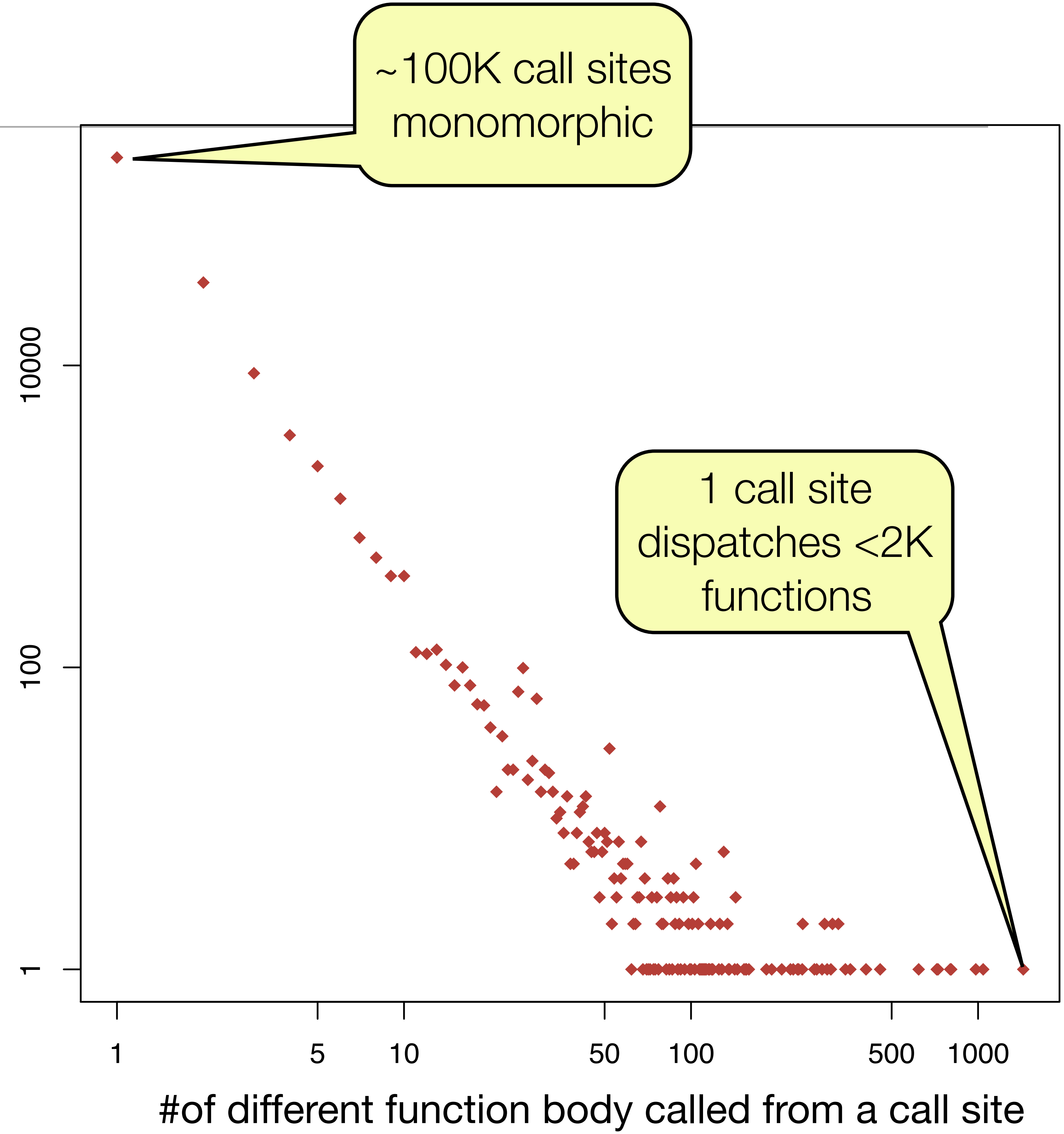
- What if there were many identical functions?
- Programming style (or lack thereof) matters




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

Dynamic dispatch

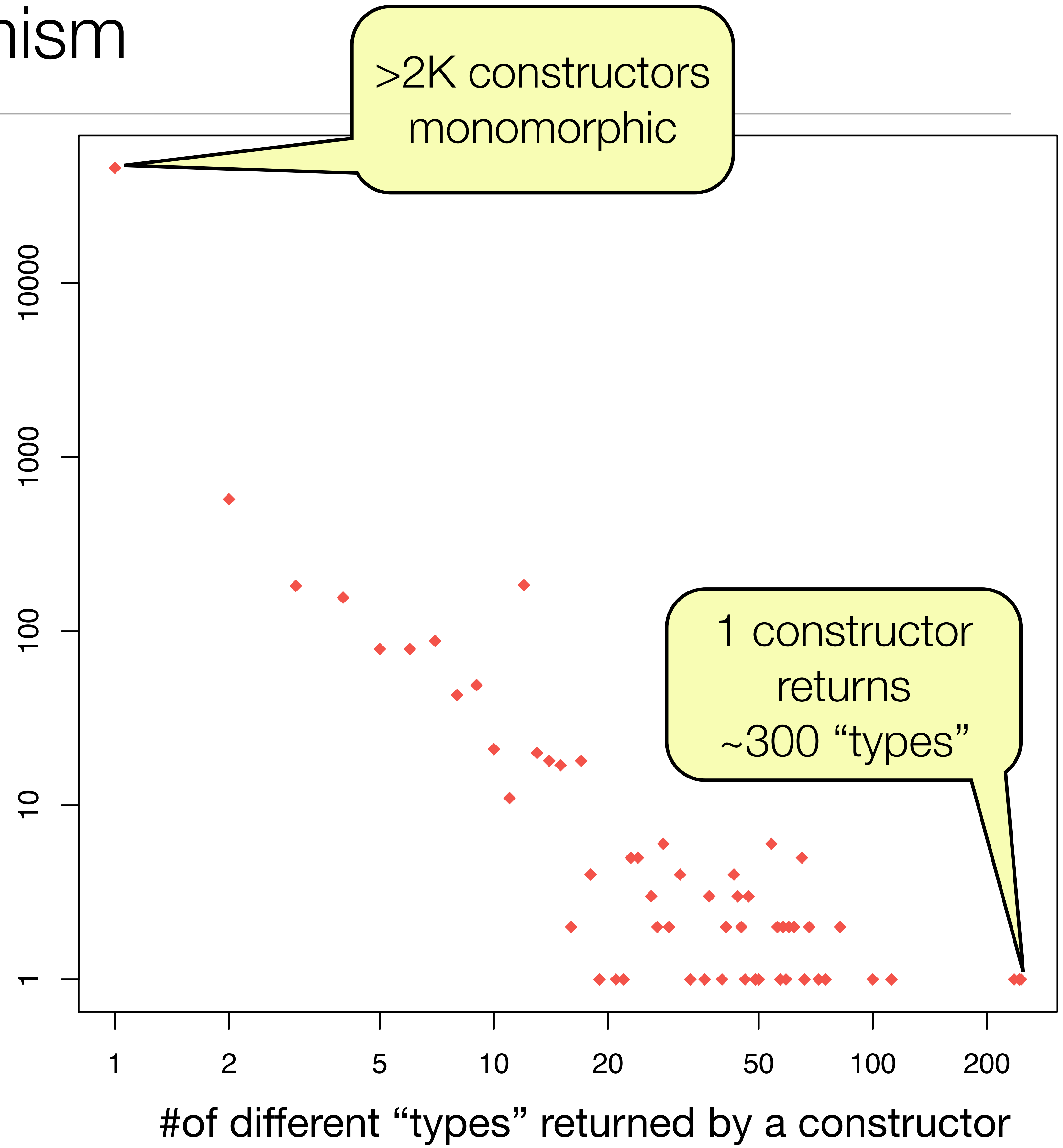
- These are not so different from Java or C#…



Constructor dynamism

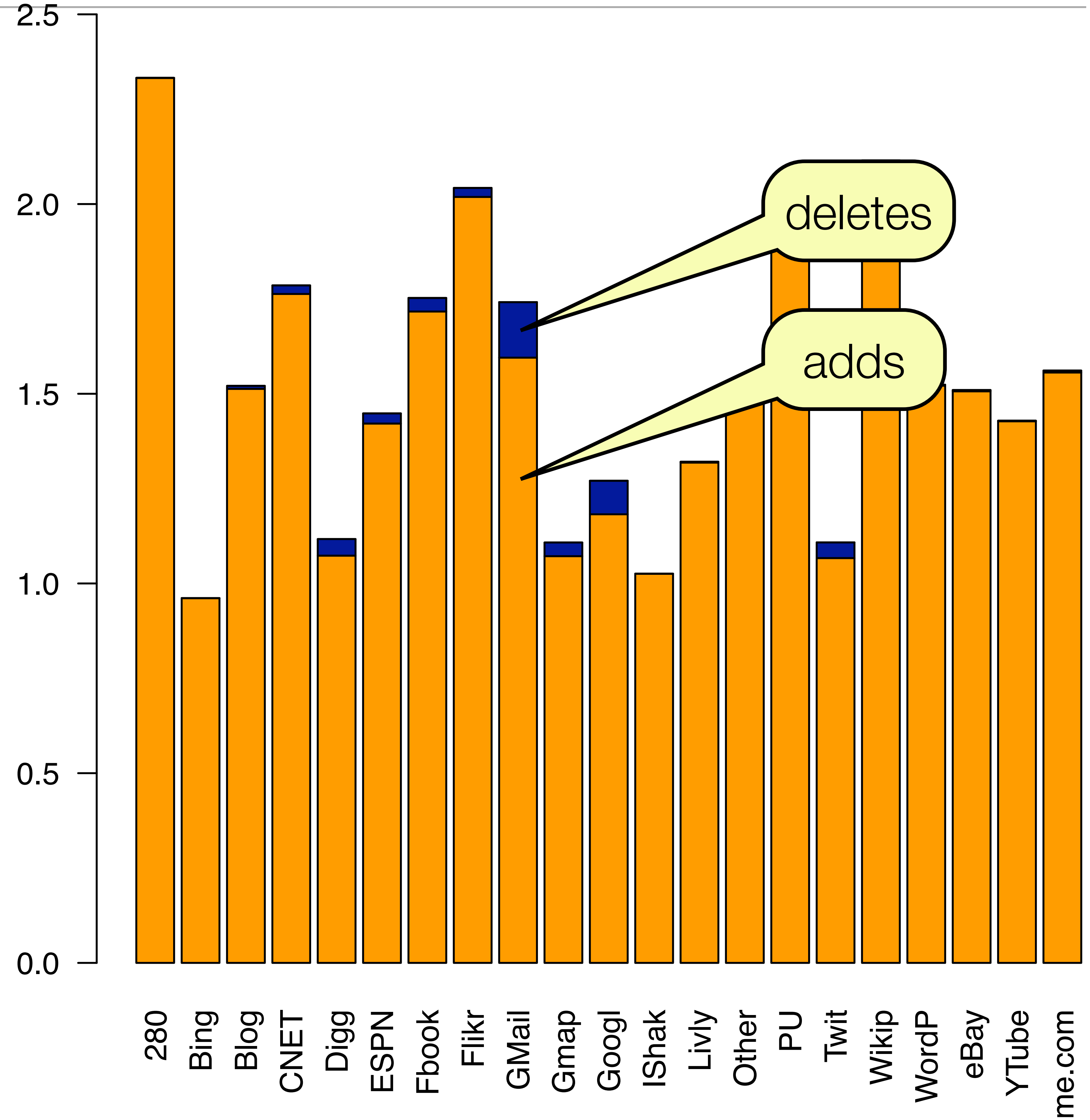
- Constructors are "just" functions that side-effect **`this`**.
- Accordingly a constructor can return different "types", i.e. objects with different properties

```
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
```

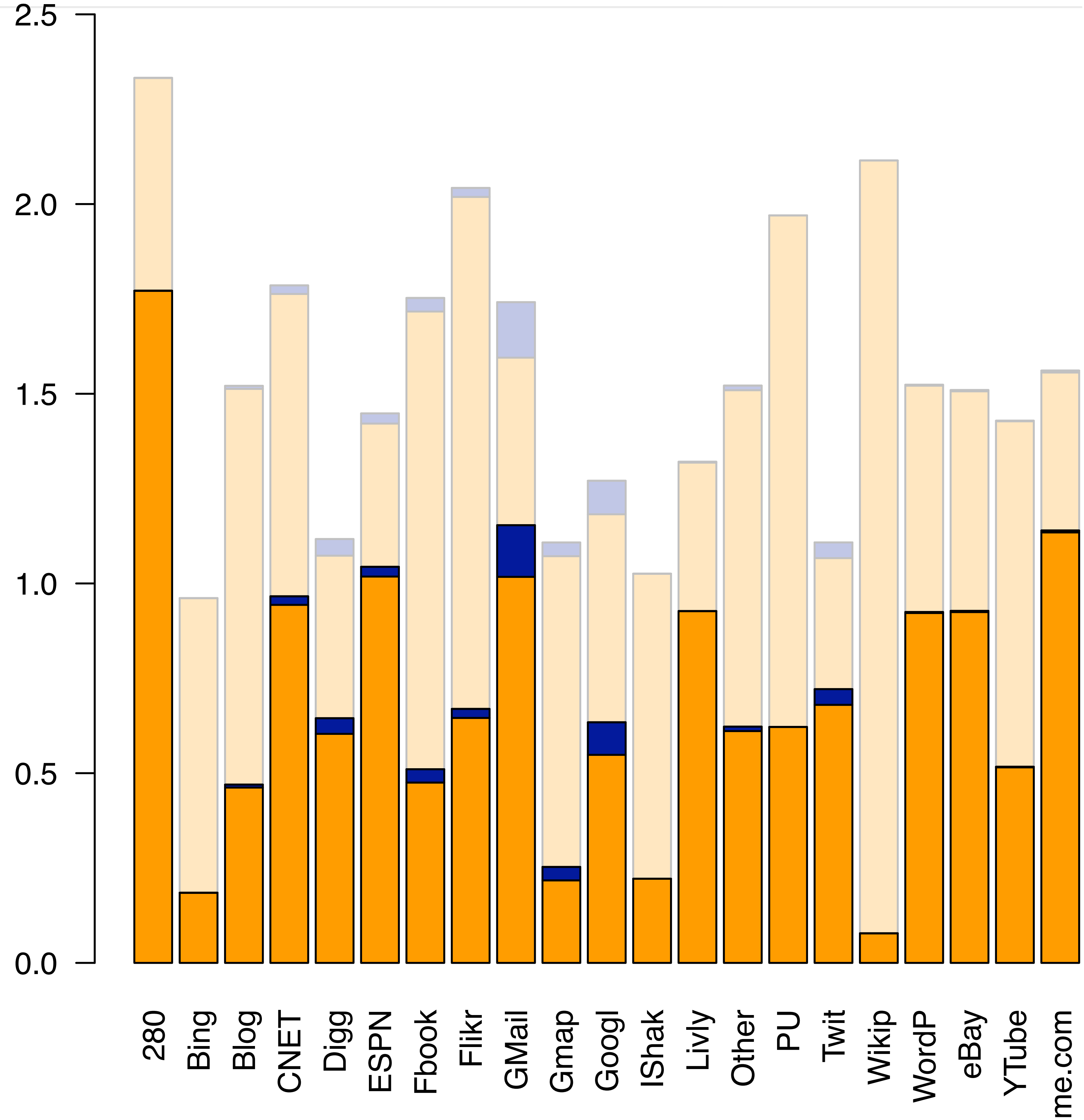
Addition/Deletion

- 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 …
```

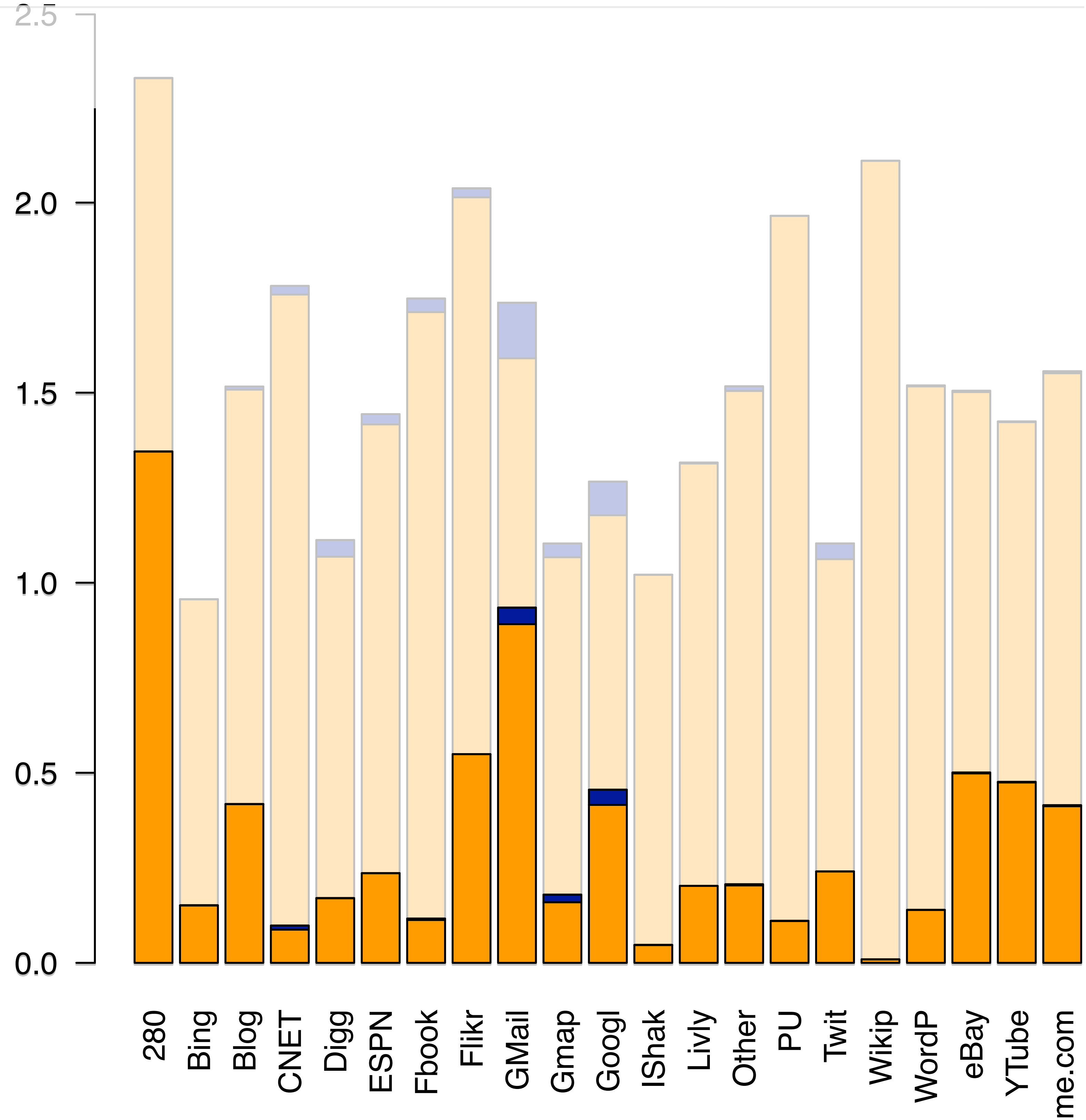


Addition/Deletion

- 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, *maybe*

- 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

Future Prospects

THE MODERN THEORY OF THE DESCENT OF MAN.

Thorn

- The story so far:
    - We are looking into how to help scripts grow up to be programs
    - Scripts are really dynamic
    - *Existing languages are ill-suited to non-invasive evolution*

Methodology

- Design a new language, *benefit from*
    - *the ability to correct unlucky language design decision*
    - *the freedom from legacy code and user base*
- Thorn is an experiment in language design
- Thorn lets scripts grow up by
    - addition of encapsulation/modularization (classes, modules, components)
    - addition of concurrency (components)
    - addition of types (like types)

Thorn is a Programming Language

- A familiar syntax

```
class List(hd,tl) {

    def lmap(f) = List( f.apply(head),
                        if(hd==null) tl.lmap(f); else null;

}


ls = List(1, List(2, List(3, null)));
```

Growing up modular

Modularity and encapsulation mechanisms

- A script
  - A simple script that does some text manipulation
  - No encapsulation, no modularity, but does the job

```
words = "story.txt".file.contents.split("\\W+");
wc = %group(word=w.toLower){ n=%count; | for w<-words };
sorted = %sort["%3d %s".format(n,word)
              %> n %< word | for {:word, n:} <- wc];
println( sorted.joined("\n") );
```

Modularity and encapsulation mechanisms

- A class
  - Classes provide basic encapsulation and modularity
  - Limited support for access control

```
class Counter(name) {

   words = name.file.contents.split("\\W+");

   def count() = %group(word=w.toLower) {
                    n=%count; | for w <- words };

   def sort() = %sort["%3d %s".format(n,w)
                      %> n %< w | for {:w,n:} <- count()];

}
```

Modularity and encapsulation mechanisms

- A module
    - Better support for encapsulation
    - Control over linking

```
module COUNT {

 import own file.*;
 import util.Vector;

 class Counter(name) {
     …
   }
}
```

Modularity and encapsulation mechanisms

- A component
  - Stronger encapsulation

```
component Count {
  sync count(name) ...
}
```

Growing up concurrent

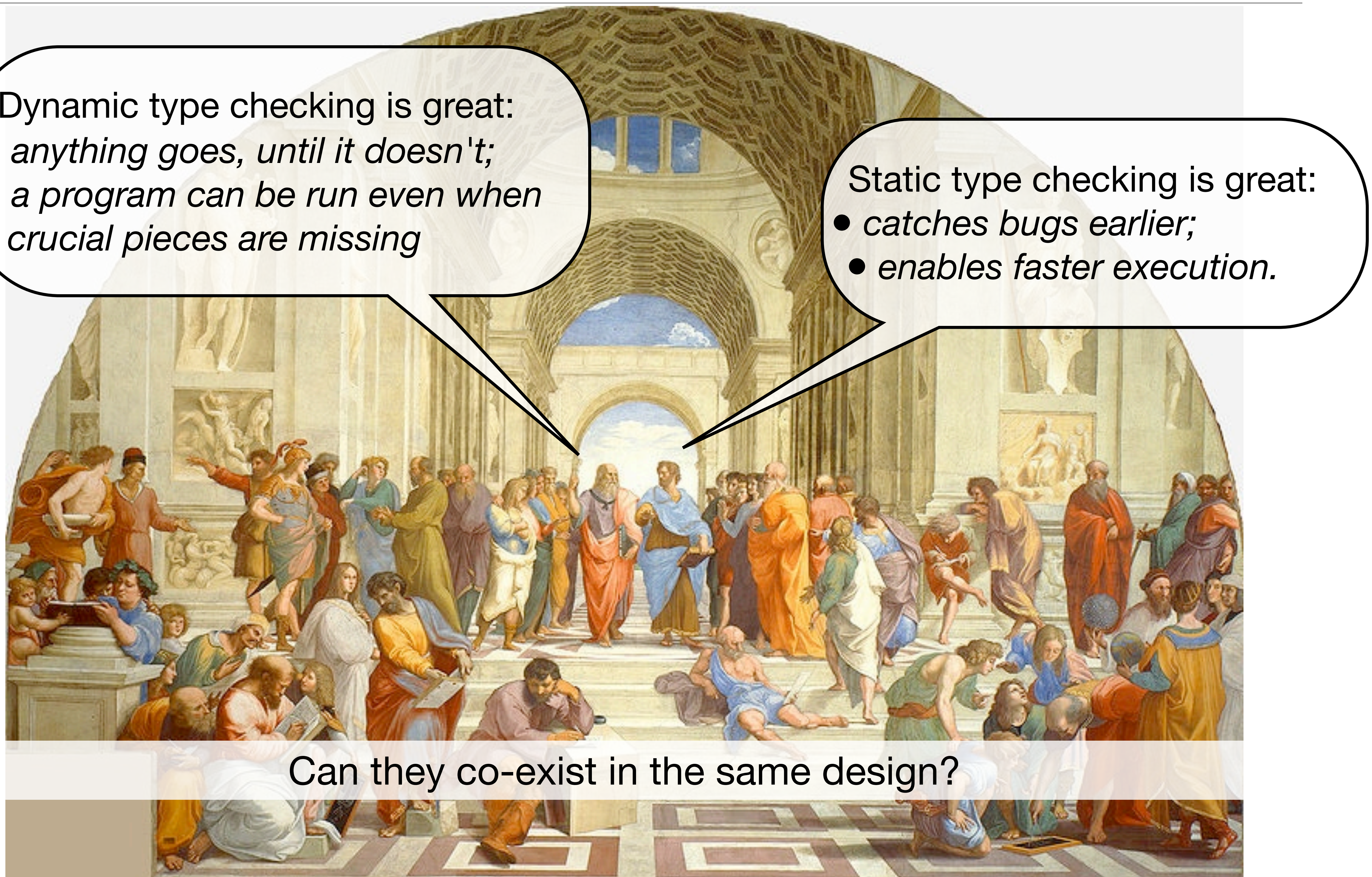
Concurrency features

- The unit of concurrency is the component
- Components are
  - single-threaded
  - fully isolated
  - have a mailbox

More details in Bard's OOPSLA talk

Growing up typed

Static and dynamic type checking



Problem

```
class Foo{ def bar(x:Int) = x+1; }

a:Foo = Foo();

a.bar(X);
```

assume no static type information available on X

Idea: let the run-time check that **X** is compatible with type `Int`.

When should this check be performed?
How long does it take?

Run-time wrappers

```
class Ordered { def compare(o:Ordered):Int; }

fun sort (x:[Ordered]):[Ordered] = …

a:[Ordered] = sort(X);
```

- Checking that **`X`** is an array of `Ordered` is linear time
- Arrays are mutable, so checking at invocation of `sort` is not enough.

Idea: add a wrapper around **`X`** that checks that it can respond to methods invoked on it

Compiled code:

```
a:[Ordered] = sort(#[Ordered]#X)
```

Cannot optimize code

```
class Foo{ def bar(x:Int) = x+1; }

a:Foo = Foo();

a.bar(X);
```

With static types, body of `Foo.bar` compiled with aggressive optimizations

Any decent compiler would unbox the `Int`

This is not possible if it is a wrapped object `#Int#X`

Our design

Our design principles

Permissive:

accept as many programs as possible

Modular:

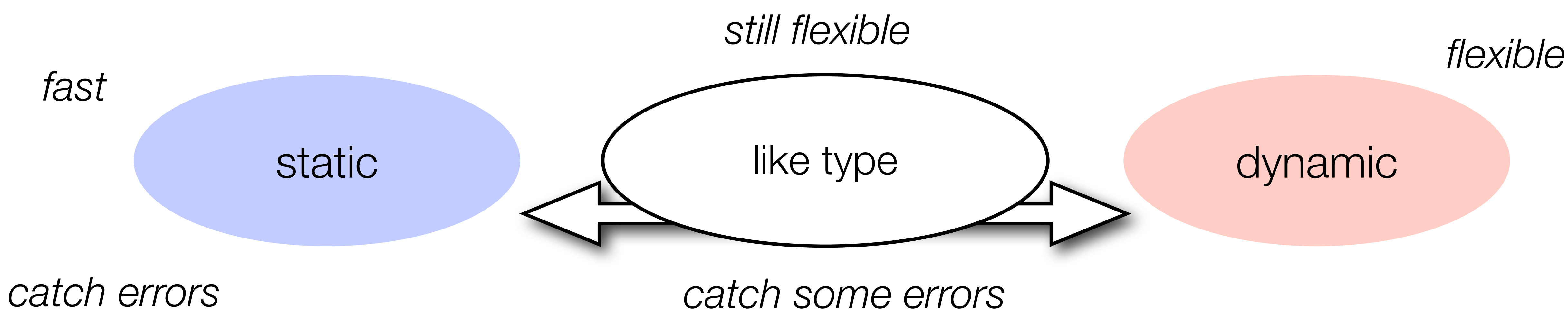
be as modular as possible

Reward good behavior:

reward programmer with performance or clear correctness guarantees

Our design

Introduce a novel type construct that mediates between static and dynamic.



- For each class name `C`, add type `like C`
- Compiler checks that operations on `like C` variables are well-typed if the referred object had type `C`
- Does not restrict binding of `like C` variables,
  checks at run-time that invoked method exists

An example

```
class Point(var x:Int, var y:Int) {
  def getX():Int = x;
  def getY():Int = y;

  def move(p           ) {  x := p.getX(); y := p.getY(); }

}
```

Requirements:

1. Fields **`x`** and **`y`** declared **`Int`**
2. **`move`** accepts any object with **`getX`** and **`getY`** methods

like Point

```
class Point(var x:Int, var y:Int) {
  def getX():Int = x;
  def getY():Int = y;

  def move(p:like Point) {  x := p.getX(); y := p.getY(); }

}
```

1. Flexibility

```
class Point(var x:Int, var y:Int) {
  def getX():Int = x;
  def getY():Int = y;

  def move(p:like Point) {  x := p.getX(); y := p.getY(); }

}

class Coordinate(x:Int,y:Int) {
  def getX():Int = x;
  def getY():Int = y;
}

p = Point(0,0);
c = Coordinate(5,6);
p.move(c);
```

`move` runs fine if `c` has `getX`/`getY`

2. Checks

```
class Point(var x:Int, var y:Int) {
  def getX():Int = x;
  def getY():Int = y;

  def move(p:like Point) {
       x := p.getX(); y := p.getY();
       p.hog;
  }
}
```

Compile-time Error

move is type-checked under assumption that the argument is a **Point**

3. Return values have the expected type

```
class Cell(var contents) {
    def get() = contents;
    def set(c) { contents := c }
}

class IntCell { def get():Int;   def set(c:Int);  }

q:Cell = Cell(41);
p:like IntCell = Cell(42);

q.get() + 1;

p.get() + 1;
```

q.get():dyn

p.get():Int
optimizations possible

Like types

- A *unilateral promise as to how a value will be treated locally*

  allow most of the regular static checking machinery

  allows the flexibility of structural subtyping

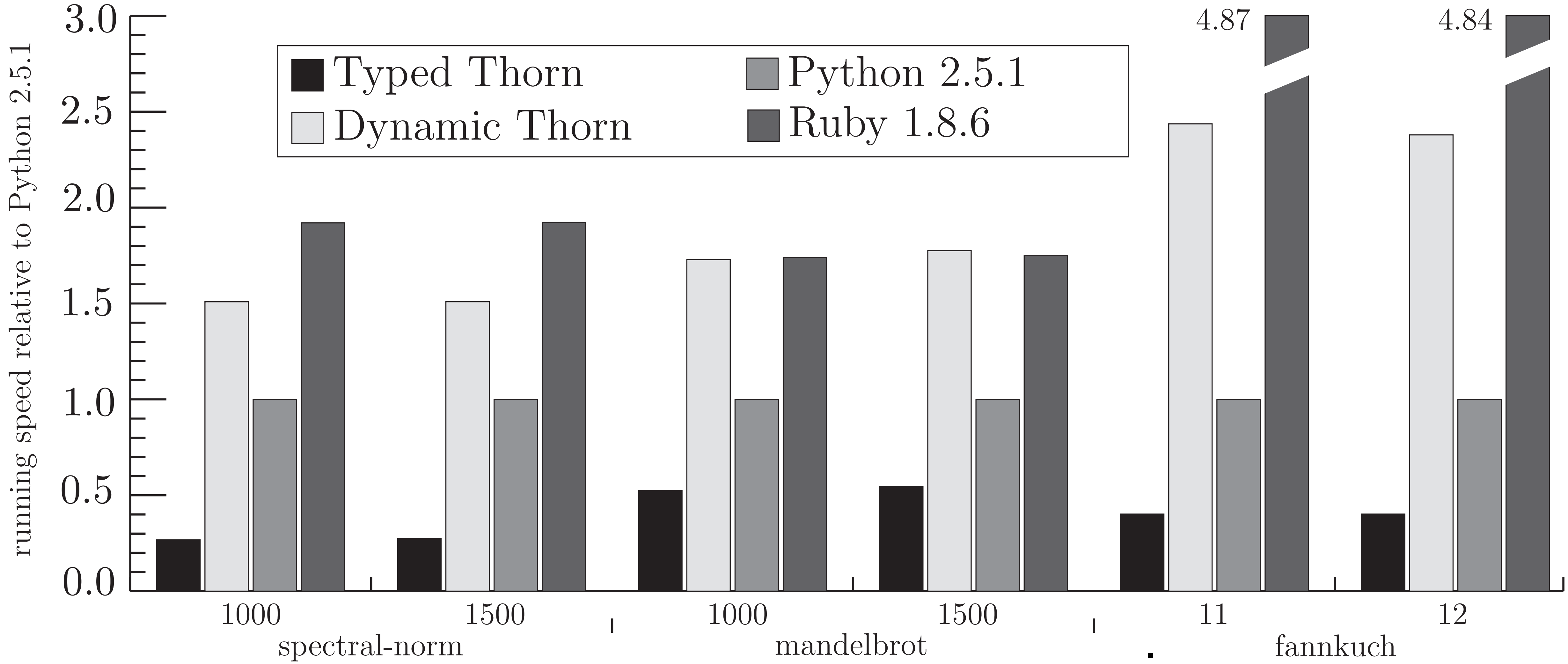
- *Concrete types can stay concrete*

  allow reusing names as semantics tags

- *Interact nicely with generics*

Rewards

Lies, big and small...



- * Obtained with an older version of Thorn
- * Benefits due to unboxing of numeric values
- * Benchmarks are meaningless
- * Still slower than Java

Related Work

- *Findler, Felleisen.* Contracts for higher-order functions. 2002
- *Bracha.* The Strongtalk Type System for Smalltalk. 2003
- *Gray, Findler, Flatt.* Fine-grained interoperability through mirrors and contracts. 2005
- *Siek, Taha.* Gradual typing for functional languages. 2006
- *Tobin-Hochstadt, Felleisen.* Interlanguage migration: From scripts to programs. 2006
- *ECMA.* Proposed ECMAScript 4th Edition – Language Overview. 2008
- *Siek, Garcia, Taha.* Exploring the design space of higher-order casts. 2009
- *Wadler, Findler.* Well-typed programs can't be blamed. 2009

Current team: Bard Bloom, John Field, Johan Ostlund, Gregor Richards, Brian Burg, Nick Kidd
Alumni: Tobias Wrigstad, Nate Nystrom, Rok Strinsa
Sponsor: IBM, ONR, NSF

Conclusion, *actually*

- Transforming a one-off script into a program requires language and virtual machine support
- Language design may have to make certain compromises in order to achieve performance
- Rewarding programmer effort is essential for adoption

- More information:

  THORN tutorial @ OOPSLA

  Talk in the main track

  Online-demo

http://thorn-lang.org