The Rise of Dynamic Languages

Jan Vitek
Programming Languages...

... provide a vocabulary for computational thinking
... are measured in the time to solution
... designs manifest tensions

end-users vs. CS
exploratory vs. batch
interpretive vs. compiled
Lisp vs. Fortran
50+ years ago...

Needless to say, the point of the exercise was not the differentiation program itself ... but rather clarification of the operations involved in symbolic computation

[McCarthy, History of Lisp HOPL’78]
Two decades of dynamism

<table>
<thead>
<tr>
<th>Language</th>
<th>Approach</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>Visual Basic</td>
<td>dyn</td>
<td>1991</td>
</tr>
<tr>
<td>Python</td>
<td>dyn</td>
<td>1991</td>
</tr>
<tr>
<td>Lua</td>
<td>dyn</td>
<td>1993</td>
</tr>
<tr>
<td>R</td>
<td>dyn</td>
<td>1993</td>
</tr>
<tr>
<td>Java</td>
<td>stat+dyn</td>
<td>1995</td>
</tr>
<tr>
<td>JavaScript</td>
<td>dyn</td>
<td>1995</td>
</tr>
<tr>
<td>Ruby</td>
<td>dyn</td>
<td>1995</td>
</tr>
<tr>
<td>C#</td>
<td>stat+dyn</td>
<td>2001</td>
</tr>
<tr>
<td>Scala</td>
<td>stat</td>
<td>2002</td>
</tr>
<tr>
<td>F#</td>
<td>stat</td>
<td>2003</td>
</tr>
<tr>
<td>Clojure</td>
<td>dyn</td>
<td>2007</td>
</tr>
<tr>
<td>Sho</td>
<td>dyn</td>
<td>2010</td>
</tr>
</tbody>
</table>
Popularity
Java
C++
VB
C#
C
SQL
JS
PHP
Perl
Python
Ada
Pascal
Asm
Cobol
Fortran
AScript
Delphi
Smalltalk
Lisp
Obj-C
ColdFus
Tcl
Shell
Scheme
Haskell
Scala
Rexx
Erlang
Forth
Ocaml

http://langpop.com
Dynamic Languages...

- Dynamic Typing
- Single threaded
- Late binding
- Failure oblivious
- Reflective
- Garbage collected
- Interactive
- Embeddable/Extendible
- Permissive
- High-level Data Structures
- Lightweight syntax
- Performance challenged
Dynamic languages are everywhere…
Dynamic languages are popular…
Dynamic languages are successful…
Dynamic languages are growing up…
case study: R

... a language for data analysis and graphics.

... widely used in statistics

... based on S by John Chambers at Bell labs

... open source effort started by Ihaka and Gentleman
Workflow based on interaction with the IDE:
- read data into variables
- make plots
- compute summaries
- more intricate modeling steps
- develop simple functions to automate analysis

…
case study: R

... vast libraries of reusable codes
... well documented and self-testing
... 4,338 R packages in CRAN and other repositories
• Functional and concise

```r
cube <- function(x=5) x*x*x
```

cube()
cube(2)
cube(x=4)
case study: R

Powerful array and matrix operations

\[
x \leftarrow c(2, 7, 9, 2, \text{NA}, 5)
\]

\[
x[1:3]
\]

\[
x[-1]
\]

\[
x[\text{is.na}(x)] \leftarrow 0
\]
case study: R

Powerful graphics

```r
> plot(cars)
> lines(lowess(cars))
```
case study: R

R is Lazy

> with(formaldehyde, carb * optden)

[1] 0.008 0.080 0.223 0.322 0.438 0.703
case study: \texttt{R}

... tools and support for reproducible experiments

Recent \textit{NYTimes} story on uncovering faulty research

case study: R

... is a dynamic language

... is a vector language

... is an object-oriented language

... is a functional language

... is a lazy language

Lightweight
Embeddable
Extendible
Failure oblivious

Single threaded
Portable
Dynamic Typing
Interactive

Reflective
High-level Data
Permissive
Open
Dynamic Typing

Dynamic languages use *Duck Typing*

"When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck."  -- JW Riley

More precisely

```java
fun F(x) {
    if (alwayFalse) x.ready()
    else x.draw()
}
```
Dynamic Typing

“In a strongly typed language each data area has a distinct type & each process states its communication requirements in terms of these types.” -- K. Jackson ’77

If static typing has benefits such as:

• preventing some errors ahead of time
• simplifying generation of efficient code
• providing machine-checked documentation

Why is it a bad idea?
Dynamic Typing

Static typing only catches trivial errors

most systems can’t even catch NPEs, or off-by-one errors

Static typing ossifies code and hinders evolution

make the type checker globally happy before testing a local change

Static typing slows down the rate of development

pessimistic typing, in case of doubt just say no
Dynamic Typing

Hypothesis:

No difference in time solving semantic bugs with a dynamically or statically typed language

Steinberg. What is the impact of static type systems on maintenance tasks? MSc Thesis U.Duisburg-Essen
case study: JavaScript

91% of top 10K web pages!

- Lightweight
- Embeddable
- Extendible
- Failure oblivious

- Single threaded
- Portable
- Dynamic Typing
- Interactive

- Reflective
- High-level Data
- Permissive
- Open
Reflection

… refers to the runtime manipulation of program structures

recall the R with keyword?

\[
> \text{with(fdehyde,} c^*o) \\
\]

It is actually a generic function:

\[
\textbf{with.default} \\
\text{<- function(data,expr,...)} \\
\text{eval(substitute(expr),} \\
\text{data,} \\
\text{enclos=parent.frame()})
\]
Reflection

Access object properties

Update object properties

Delete object properties

Discover properties

Access global variables

Update global variables

Access/update local variables

```
x["f"]
x["f"] = 2
delete x.f
for(var p in x){
  window["f"]
}
window["f"] = 2
eval("f = 2")
```
case study: Lua

- Used in the gaming industry
- C library for seamless embedding
- Interoperation requires reflection over data

Lightweight  Single threaded  Reflective
Embeddable   Portable     High-level Data
Extendible    Dynamic Typing Permissive
Failure oblivious Interactive    Garbage-collected

Lerusalimschy, et. al. *Passing a Language through the Eye of a Needle*, ACMQUEUE, 2011
case study: Lua

Adobe Lightroom

Used ...

… to provide interface and glue between components

… for business logic, controllers, views

… for its fast turn around

An embeddable language must be have an API that allows data to be accessed and manipulated externally.

- JavaScript designed to be embedded in HTML pages
- Interaction with browser adds “isolation”-based security model
- Document Object Model exposes the web page
case study: Embeddable

Mercury

Python

#Call at each cycle of Mercury execution
energyTal = mc.tally.tal
["EnergyDeposition"]

if energyTal.getValue(Particle="Neutron", Cell="Skull") > 1e-6:
    print "Neutron energy deposition to the skull reached threshold."

... C++ parallel Monte Carlo particle transport code
... embeds Python to ease testing & validation
... massively faster development cycle

case study: Extendible

... inertial confinement fusion simulation
... extends C++ to provide a "steerable" simulation
... wrapped and exposed to Python via SWIG
... 1.7Mloc generated C++ wrappers (static price)

---

Alumbaugh, Dynamic Languages for HPC at LLNL. Talk at VEESC Workshop, 2010
Failure Obliviousness

Dynamic languages keep the program running…

… by allowing the execution of incomplete programs

… by converting data types automatically when possible

… by decreasing number of errors that must be handled

“Best effort” execution
Failure Obliviousness

Getting an error in JavaScript is difficult

```javascript
x = {}; // object
x.b = 42; // field add
y = x["f"]; // undefined
z = y.f; // error
```
case study: CERN

- Dynamic languages used: Python, Perl, Bash, Tcl, ...
- But, most of the analysis code is in C++

Can C++ be turned into a dynamic language?

- Lightweight
- Embeddable
- Extendible
- Failure-oblivious
- Single-threaded
- Portable
- Dynamic Typing
- Interactive
- Reflective
- High-level Data
- Permissive
- Open
case study: CERN & CINT

• From 1991, 400KLOC; parser, interpreter, reflection

• Interface to ROOT data analysis framework, >20k users

Ideally:

- Higher level syntax
- Faster
- Threading

```cpp
foreach electron in tree.Electrons

vector<Electron>* ve = 0;
tree->SetBranchAddress("Electrons", ve);
for (int i=0; i<ve.size(); ++i) {
    Electron* electron = ve[i];
}
```

Antcheva, Ballintijn, Bellenot, Biskup, Brun, Buncic, Canal, Casadei, Couet, Fine, Franco, Ganis, Gheata, Gonzalez Maline, Goto, Iwaszkiewicz, Kreshuk, Segura, Maunder; Moneta, Naumann, Offer, Onuchin, Panacek, Rademakers, Russo, Tadel.

case study:

Pluto

... manages the retirement savings of 5.5 million users

... for a value of 23 billion Euros

320 000 lines of Perl
68 000 lines of SQL
27 000 lines of shell
26 000 lines of HTML

Lundborg, Lemonnier. PPM or how a system written in Perl can juggle with billions. Freenix 2006
case study: Perl

High productivity: *Perl wins over Java*

Disciplined use of the language: *Many features disallowed*

Home-brewed contract notation: *Runtime checked*

Lightweight

Embeddable

Extendible

Failure oblivious

Single threaded

Portable

Dynamic Typing

Interactive

Reflective

High-level Data

Permissive

Open
Performance
- 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

© Nate Nystrom, '07
Startup times

- C interpreters - 4-20x faster than Java
- Java interpreters - 5-6x slower than Java

Hello time / Java

© Nate Nystrom, '07
Performance

Does not matter for…

... short running and I/O bound codes

But, when performance matters…

... rewrite applications in C and lose benefits of dynamism
Conclusion

Dynamic languages increase the velocity of science

Dynamic languages are a gateway drug to computing

Dynamic languages need some static features, some of the time
Research Challenges

Can dynamic languages enjoy the correctness and efficiency of static languages, while remaining dynamic?

- Understanding dynamism in the wild
- Tracing-JITs: Highly-optimized adaptive compilation
- Gradual types and other incremental static type systems
- Capturing more expressive invariants with Code Contracts
how dynamic is dynamic

• A familiar syntax

```javascript
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; });
```
methodology
Corpus

Traced Alexa top 100 sites
Multiple traces per site
8GB of trace data
500MB distilled database

<table>
<thead>
<tr>
<th>Alias</th>
<th>Library</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>280S</td>
<td>Objective-J¹</td>
<td>280slides.com</td>
</tr>
<tr>
<td>BING</td>
<td></td>
<td>bing.com</td>
</tr>
<tr>
<td>BLOG</td>
<td></td>
<td>blogger.com</td>
</tr>
<tr>
<td>DIGG</td>
<td></td>
<td>digg.com</td>
</tr>
<tr>
<td>EBAY</td>
<td></td>
<td>ebay.com</td>
</tr>
<tr>
<td>FBOK</td>
<td></td>
<td>facebook.com</td>
</tr>
<tr>
<td>FLKR</td>
<td></td>
<td>flickr.com</td>
</tr>
<tr>
<td>GMAP</td>
<td></td>
<td>maps.google.com</td>
</tr>
<tr>
<td>GMIL</td>
<td></td>
<td>gmail.com</td>
</tr>
<tr>
<td>GOGL</td>
<td></td>
<td>google.com</td>
</tr>
<tr>
<td>ISHK</td>
<td></td>
<td>imageshack.us</td>
</tr>
<tr>
<td>LIVE</td>
<td></td>
<td>research.sun.com/me.com</td>
</tr>
<tr>
<td>MECM</td>
<td></td>
<td>twitter.com</td>
</tr>
<tr>
<td>TWIT</td>
<td></td>
<td>wikipedia.com</td>
</tr>
<tr>
<td>WIKI</td>
<td></td>
<td>wordpress.com</td>
</tr>
<tr>
<td>WORD</td>
<td></td>
<td>youtube.com</td>
</tr>
<tr>
<td>YTUB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALL</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

¹Objective-J
²jQuery
³Closure
⁴Prototype
⁵SproutCore

Average over 103 sites
assumptions
1. Program size Modest
2. Call-site dynamism Low
3. Function signatures Meaningful
4. Properties added at initialization
5. `eval` infrequent and harmless
Program Size is Modest
Program Size is Modest

Size of source in bytes

![Bar chart showing the size of source code for various websites, with 280slides being the largest and eBay, me.com, and YouTube having smaller sizes.](chart.png)
Call-site Dynamism is Low
of different function body called from a call site

1 call site dispatches >1K functions

~100K call sites monomorphic
Properties are Added at Object Initialization
Google

Objects are dead
Function Signatures are Meaningful
function Person(n,M)
    this.name=n;
    this.sex=M;
    if(M)
        this.likes= "guns"
}
eval is Infrequent and Harmless

Richards, Hammer, Burg, Vitek. The Eval that Men Do: A Large-scale Study of the Use of Eval in JavaScript Applications. ECOOP 2011
<table>
<thead>
<tr>
<th></th>
<th>JavaScript used</th>
<th>eval use</th>
<th>Avg eval (bytes)</th>
<th>Avg eval calls</th>
<th>total eval calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTERACTIVE</td>
<td>100%</td>
<td>59%</td>
<td>1486</td>
<td>38</td>
<td>2,434</td>
</tr>
<tr>
<td>RANDOM</td>
<td>91%</td>
<td>43%</td>
<td>687</td>
<td>85</td>
<td>367,544</td>
</tr>
</tbody>
</table>

**Figure: Distribution of Sites, Calls, and Strings**

- **Sites:**
  - Interactive: 77
  - Random: 1331

- **Calls:**
  - Interactive: 3491
  - Random: 111535

- **Strings:**
  - Interactive: 230387
  - Random: 527529
Categories of eval

- **JSON**: A JSON string or a variant.
- **JSONP**: A padded JSON string.
- **Library**: One or more function definitions.
- **Read**: Read access to an object's property.
- **Assign**: Assignment to a local variable or an object property.
- **Typeof**: Type test expression.
- **Try**: Trivial try-catch block.
- **Call**: Simple function/method call.
- **Empty**: Empty or blank string.
- **Other**: Uncategorized string.

Examples:
- `eval('{x: 2}')`
- `eval('obj.f')`
- `eval('id = {x: 2}')`
- `eval('typeof(z_+y[i]+')!=''undefined''')`
- `eval('document.getElementById("m")')`
Fig. 8. Patterns by websites.

Number of web sites in each data set with at least one eval argument in each category va single web site can appear in multiple categoriesw2

Fig. 9. Patterns.

Ratio of evals in each category2

Both JSON and JSONP are quite common2 In each data setz JSONP is at worst the third most common category in both Fig2 (and Fig2)

z and JSON and JSONP strings accounted for between 77s vR

RANDOM w and 8-s vI

INTERACTIVE w of all strings eval’d2

Since most call sites do not change categories vdiscussed later in Section 2.2. these numbers indicate that analyses could make optimistic assumptions about the use of eval for JSONz but will need to accomodate the common pattern of JSON being assigned to a singlez often easilyfideterminablez variable2

Most of the remaining evals are in the categories of simple accesses2 Property and variable accessesz both simple accesses which generally have no sidefieffectsz are in all data sets amongst the second to fifth most common categories for sites to use2 They account for (sz 7-s and 79s of eval calls in I

INTERACTIVE zP

AGE L

OAD R

RANDOM z respectively2 The most problematic categories - appear in fewer sitesz but seem to be used frequently in those sites where they do appear2 Howeverz this does not include uncategorized evalsz which also have problematic and unpredictable behavior2

Impact on analysis.

Most eval call sites in categories other than Library z Other and Call are replaceable by less dynamic features such as JSON.parsez hashmap accessz and proper use of JavaScript arrays2 On I

INTERACTIVE z these categories account for -

By problematic categoriesz we include evals with complex side effects such as assignments and declarationsz and those categories with unconstrained behavior such as calls2
The distribution of provenance is significantly different for the P
composites of only constants and around 6zz strings were just a constant in the sourcex
provenance of strings was
each call to
typesv which is not surprisingx Figx
syntaxv so it is unsurprising that they are generated from constant stringsx
other data setsx Many of these are often misused as replacements for arrays or hashmap
eval
would be forbiddensx Many major web sites have a similar separation of contentx
circumvent the sameworigin policy runder which the straightforward AJAX approach
stores its JavaScript on a separate subdomainv this convoluted pattern is necessary to
tigation into the low proportion of
brary
Provenance vs. Patterns
R
ANDOM
The I
INTERACTIVE
NTERACTIVE
Data setsx For thesev
patternsx Upon further invesw
is virtually nonexistentx
provenancev we found thatv

The figure below illustrates the proportion of strings with given provenance:

- **Input**: 50% of strings are input from the user.
- **AJAX**: 30% of strings are retrieved from an AJAX request.
- **DOM**: 15% of strings are DOM manipulations.
- **Composite**: 5% of strings are composite inputs.
- **Constant**: 0% of strings are constant values.

For example, in the 'DOM' category, the dominant
Composite and Input categories are used in equal proportion, while
Assign and Read categories are used with similar proportions in the
AJAX, Random, and Synthetic data sets. For the Interactive data set, the DOM
source was virtually nonexistent. For the Random data set, the provenance
dominates all others.
Industry Benchmarks are Representative

- Benchmarks (SunSpider, V8...) drive implementations
- Results are useful, if they reflect real programs
This suggests that eval performs no...
Does it matter?

![Graph comparing performance improvements in different versions of Firefox](image)

- Amazon 9
- Sunspider 0.9.1

Trace - based Compilation
Basic idea...

- If programs repeatedly take the same path, compile and optimize *that* path

- The basic algorithm discovers stable paths by:
  1. executing in interpreted mode and recording path information
  2. at anchor points, compile hot path and switch to compiled code
  3. detect path hazards with dynamic guards

```javascript
var sum = 0
for (var i = 0; i < 1000; i++) {
    if (i == 990) {
        sum += " Hello World "
    }
    sum += 1
}
print(sum)

// result: "989 Hello World 11111111111"
```javascript
var sum = 0
for (var i = 0; i < 1000; i++) {
    if (i == 990) {
        sum += " Hello World 
    }
    sum += 1
}
print(sum)
```

"+" depend on operands, leading to lots of runtime checks

Optimize loop for an integer ‘‘+’’

After "+" becomes a string, resume in un-optimized code
```javascript
var sum = 0
for (var i = 0; i < 1000; i++) {
    if (i == 990) {
        sum += " Hello World "
    }
    sum += 1
}
print(sum)
```
We go from this ...
10 loop instructions, 2 loop guards!

Performance 7x faster than the CLR based JScript, and slightly faster than V8
Static and dynamic type checking

Dynamic type checking is great:
- anything goes, until it doesn't;
- a program can be run even when crucial pieces are missing

Static type checking is great:
- catches bugs earlier;
- enables faster execution.

Can they co-exist in the same design?
Problem

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

a: Foo = Foo();
a.bar(Y);
# assume no static type information available on Y
```

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

*When should this check be performed? How long does it take?*
Run-time wrappers

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

fun sort (x:[Ordered]):[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:

`sort(#[Ordered]#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
Introduce a novel type construct that mediates between static and dynamic.

- For each class name $C$, add type $\text{like } C$
- Compiler checks that operations on $\text{like } C$ variables are well-typed if the referred object had type $C$
- Does not restrict binding of $\text{like } C$ 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
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(); }
}

Flexibility

```scala
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);
move runs fine if c has getX/getY
p.move(c);
```
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;
    }
}
Rewards ...
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

Flanagan. ValleyScript: It’s like static typing. 2007

Tobin-Hochstadt, Felleisen. Interlanguage migration: From scripts to programs. 2006

Wadler, Findler. Well-typed programs can’t be blamed. 2009
Code Contracts
Contracts

• Precondition
  What I expect from the caller?  e.g. non-null parameter

• Postcondition
  What I ensure to the caller?  e.g. value is non-negative

• Object Invariant
  What holds in the stable states of an object?  e.g. field non-null
Specify ...

```
T Pop() {
    return this.a[--next];
}
```
Use code to specify code...

```csharp
T Pop()
{
    Contract.Requires(!this.isEmpty);
    Contract.Ensures(Contract.Result<T>(())!= null);
    return this.a[--next];
}
```
CodeContracts

• Language agnostic
  Write contracts in your favorite dynamic language

• Compiler transparent
  Use your usual compiler

• Leverage IDE support
  Intellisense, squiggles, debugger …

• Runtime checker enables…
  Checking postconditions
  Contract inheritance, and contract interfaces

Fähndrich, Barnett, Logozzo: Embedded contract languages. SAC'10

Fahndrich, Logozzo. Clousot: Static Contract Checking with Abstract Interpretation. FoVeOOS’10
Conclusion

• Understanding the nature of dynamic program is essential to research in the field

• Dynamic languages can match statically compiled languages if we take advantage of adaptive techniques

• Dynamic languages need to be able to assert static properties such as types and invariants