Dynamic Languages
Strike Back

Steve Yegge
Stanford EE Dept Computer Systems Colloquium
May 7, 2008
What is this talk about?

- Popular opinion of dynamic languages:
  - Unfixably slow
  - Not possible to create IDE-quality tools
  - Maintenance traps at millions of LOC
- Is the popular opinion accurate?
- We’ll look at the technology and see...
What do I mean by “dynamic language”?

- Eval, late-binding, runtime loading, mutable types, flexible dynamic dispatch, ...

- Intentionally blurring dynamic typing and dynamic features for this talk!

- Hence: Perl, Python, Ruby, JavaScript, Lisp, Scheme, SmallTalk, Lua, Tcl...
But dynamism $\neq$ type tags (or lack thereof)!

- It’s true: statically typed languages usually have some dynamic features.
- Underlying problem is cultural: people think dynamic $\Rightarrow$ dynamic typing $\Rightarrow$ slow, bad tools.
- Observation: techniques for creating tools for dynamic languages are similar to those for improving performance.
So... why do we have dynamic languages?

Stanford PhD candidate: “I don’t know why we have other langs. You only need C/C++.”

Well-known advantages to dynamic languages:
- Productivity, expressiveness, flexibility, ...

Perceived downsides: speed, tools, and the ever-elusive “maintainability”
Why are dynamic languages “slow”?

- Hard to compile with traditional techniques
- Object & variable types can change
- Methods can be added/removed
- Target machine feature mismatches
- Lack of effort: “scripting languages” are I/O bound and haven’t needed blinding speed
How can you speed up a dynamic language?

- **Language-level improvements:**
  - Native threads, optional type system, ...

- **Virtual machine improvements:**
  - Generational GC, special async I/O ops, ...

- Smarter compilers!
Historical successes

- Common Lisp: native compilers, C-like speed
- StrongTalk: static types for SmallTalk
- Scheme: cross-compile into C & use GCC
- Self: type-feedback adaptive compilers

Problem: they all sucked at marketing
Languages are no longer changing every 10 years

- Barrier to entry has gone up since 1994
- Marketing obstacles (vs. Sun, Microsoft)
- Bar has gone up for tools & infrastructure
- Open source yielded lots of useful code

Implication: we're stuck with what we've got
Pigs' attempts to fly

- Perl, Python: vanilla bytecode interpreters
- Ruby: interprets AST directly! (very slow)
- All: no usable concurrency options
- All: reference-count or mark-and-sweep GC

Java proved pigs can reach interstellar space!
Intermission/Recap

- Yesterday’s dynamic languages had great performance and great tools
- Today’s dynamic languages: not so much
- Why aren’t (more) people working to fix it?
- Ignorance, FUD and despair: “not fixable!”
- CS education failure: compilers courses!
Modern IDE expectations: autocomplete, jump-to-declaration, browsing, refactoring

IntelliJ IDEA/JavaScript: autocomplete, jump-to-declaration, browse, refactoring, ... 

What’s missing? Not much!

Java IDEs showed the way

dynamic languages now playing catch-up
Tools: Syntax

A language’s syntax yields many static clues exploitable by IDEs. Consider:

// what is the type of foo?
function foo(a, b) { return a + b; }

var bar = 17.6;  // what is bar’s type?

var x = {a: “hi”, b: “there”};  // type of x?
Tools: Domain knowledge

IDEs need to look for common idioms:

```javascript
function foo() {...}
var foo = function() {...}
foo = {a: function() {...}, b: function() {...}}
foo.prototype.x = function() {...}
with (foo) { x = function() {...} }
```

Lots of work, but no more than doing Java name and type resolution
Tools: Inference

```javascript
var foo = new Object();
var x = foo;
// how to determine that x.bar is foo.bar?
x.bar = function() {...};
```

Alias inference is similar to flow-analysis
In general: undecidable. In practice: 95+% Java IDEs also miss the ~5% reflection cases
Tools: Simulation/Emulation

- Common Java user complaint: dynamic IDEs need to run your program to be accurate
- “Not feasible to load all the code!”
- But Java runtime systems have monitoring, health checks, logging, dashboards, profiling...
- Notion that IDE “must be” separate from runtime is inaccurate in real-world scenarios
Dynamic tools: Summary

- Not harder to build than tools for static languages -- just different.
- Fundamental observation: most “dynamic” code isn’t all that dynamic
- Static analysis often possible
- Bridge gap by running/simulating the code
Performance!

- Programmers bad at tedious automation but still prefer to hand-optimize code!
- Compilers/VMs continue to get smarter
- perf “tricks” keep getting obsoleted
- Cultural problem: micro-optimization requires less thought than actual design
Micro- vs. Global-

- Walter Bright: D slower than C++, but D programs faster than C++ programs
- Java: slower than C++ in benchmarks, often faster overall (esp. with multicore)
- Ruby on Rails: 20% faster than Struts, even though Ruby is way slower than Java

Global optimizations always trump benchmarks!
Then are dynamic languages “fast enough”?

- Depends who you ask, and how you measure performance.
- Many big systems in dynamic languages: Amazon.com, Yahoo, Orbitz, NYSE, ...
- There’s still value in improving performance:
  - Browser client apps increasingly complex
  - Server farms benefit from tiny perf gains
Case Study: JavaScript

At a glance:

- Java-like syntax, prototype-based OOP
- lexical scoping, 1st-class functions, closures
- EcmaScript Edition 4: optional types
- Ajax caused surprise popularity surge
- Sudden focus on improving performance
Trick #1: classic static type inference

- var x = 0; for (i=0; i<10; i++) x += i;

- sometimes possible to infer primitive ops and generate efficient machine code

Problem: overflow changes type to Double (in JavaScript)
JIT compilation (2 of 5)

- Trick #2: Polymorphic Inline Caches (PICs)
  - Developed at Stanford (Urs Hoelzle)
  - permits inlining of polymorphic functions
  - count receiver types at call sites
  - make predictions from runtime counts
  - 50% to 100% speedup of real-world code
Trick #3: double-dispatch type inference

“box” constants with virtual interfaces

invoke operations like a+b in both directions (1st time): b.add(a), a.add(b)

now you know exact types for variables

inside loops, operands usually same type
JIT compilation (4 of 5)

Trick #4: Trace trees targeted at loops, not methods!

build up tree of runtime-compiled paths
1 path per operand type from same source
result: massive basic block fall-through

20x speedups, and can be done in O(n) time!
reports of 750x less time spent compiling
JIT compilation (5 of 5)

- Last trick for today: Escape analysis
- Statically determine whether loop values “escape” the loop (used before or after)
- If not, can optimize away object allocations (including trace boxes)
- Can save thousands of allocations in a single loop
JIT compilation: Recap

- How many of these tricks are there? Many!
- Underlying themes:
  - Most CPU consumed in loop execution
  - Runtime analysis yields smarter decisions
  - Theoretical performance exceeds C++/static
  - It’s just a lot of work that few people do
Is JavaScript “fast” yet?

- Hard to measure; benchmarks controversial
- pure-JavaScript apps beginning to compete with the desktop
- HotRuby: Ruby VM in JS - 2x-5x faster!?
- Still tons of low-hanging fruit
- Trace trees, more JIT research, ES4, ...
Beyond perf & tools

- If we solve perf and tools, what’s left?
  - Cranky programmers, ignorance, FUD
  - “maintainability” – the ultimate FUD tool
  - only solution: marketing, and lots of it
- Still several years of work left on perf/tools
- Static langs here for foreseeable future!
What have we learned?

- We're stuck with today's popular languages
- Micro-optimization best done by software
- Recent dynamic language compilation revival
- Tools/performance very possible, lots of work
- Nothing matters without marketing!
Q&A