Why Ruby isn't slow

Alex Gaynor

StrangeLoop 2013

Hi everyone. I'm really excited to be here, and I'm very excited about this topic.

About me

- Rackspace Software Engineer
- Python Software Foundation director
- Lots of Open Source stuff

So a tiny bit of background. I work for Rackspace, as a person to programs computers. I serveron the board of directors of the Python Software Foundation. I also do a lot of open source stuff. I'm also a large producer of typos and computers are terrible rants.

“Ruby is slow”

So that’s a thing people say. It’s not a particularly precise statement. Since this talk is sort of built on the premise of attacking this statement, I want to unbox what I think people mean when they say it. I also want to emphasize that the stuff I say, really applies to many dynamic languages, Python, javascript, etc. not just ruby.

Code written in Ruby executes CPU bound tasks more slowly than other languages

Friday, September 20, 13

When people say "Ruby is slow", this is usually, approximately what they're thinking. So key points: slowness is somewhat obviously relative to other languages. And we're concerned with CPU bound code. They're often implicitly substituting "MRI", Matz Ruby Interpreter, for "Ruby". And sometimes they're also thinking about parallelism.

Bad responses

So, when you ask a person, “Why do you use Ruby even though it’s slow”, you get a bunch of answers back. Sometimes people think this addresses the “Ruby is slow” thing, when instead they’re just excuses.

Bad responses

- "Our app is IO bound"

Turns out people have no idea what IO bound means. Because there's a great correlation between people who say this and apps I speed up by 30% by migrating to PyPy.

Bad responses

- “Our app is IO bound”
- “We make it up with programmer productivity”

Dynamic languages being more productive than many popular statically typed languages is probably true. It has nothing to do with performance, a total red herring, what you really meant to say is “I just don’t care” or maybe “It’s fast enough”

Bad responses

- “Our app is IO bound”
- “We make it up with programmer productivity”
- “If we need to make it fast we’ll just rewrite it {C, Scala, Java, SML}”

This is the one that makes me cry myself to sleep at night. As I’m going to explain there’s no reason dynamic languages need to be slow, and people seem hell bent on ignoring why their code is actually slow

Ruby can be fast

So, I hope it's pretty clear, I want to factually address the claim that Ruby is necessarily slow. And to do that I want to break down the myths around why Ruby, and really all dynamic languages, are slow.

THE COMPILER DOESNT KNOW THE TYPES THEREFORE IT IS SLOW

So when you ask people why dynamic languages are slow, this is usually what they say. They might also mention threads or GC, or interpreter overgead. But this is the first they say. No one knows what this means. “The compiler doesn’t know the types, so what? So it can’t optimize. Why can’t it optimize? Because it doesn’t know the types.”

Consequences

Friday, September 20, 13

So what are the actual consequences of not knowing the types?

Consequences

- All function calls are indirect

So if you were a C programmer you'd be freaking out because this means you've got JMPs which aren't well predicted and so you're getting pipeline flushes. That's cute. In most interpreters like MRI what this means is you're doing a ton of hash table lookups. Hash tables are slow.

Consequences

- All function calls are indirect
- All containers are of "Object"

Instead of nice, compact arrays of floats, you're getting an array of pointers to structs which contain floats.

Consequences

- All function calls are indirect
- All containers are of “Object”
- Instance variable lookups aren’t fixed memory offsets

Finally, as anyone who’s looked at a disassembly of a C program knows, reading a field out of a struct is just doing some magic addressing with offsets in x86. Ruby instance variables, by contrast, are often implemented on a hash table. A big slow hash table.

Let’s design a fast Ruby

So, let’s design a fast Ruby. Somethign that addresses these problems, that makes containers efficient, that makes function calls and instance variable lookups not be tons of hash tables.

RPython

So, the tool we're going to use to do this is RPython, you may have heard of it. RPython is a programming language

RPython

- Statically typed + type inference

RPython

- Statically typed + type inference
- Garbage collected

RPython

- Statically typed + type inference
- Garbage collected
- Syntax is the same as Python

So we have this language that looks like Python. Why would we use it? I have to tell you, were it just these details: the answer is never. RPython has crappy error messages, bizarre semantics, and generally atrocious UI. If you just want a type-infereced, GC'd language, there are lots of good ones, go use one. But it has one saving grace.

RPython

- JIT compiler generator

RPython, in addition to being a crappy programming language, is a framework for implementing dynamic languages. And this framework includes a “JIT generator”. Instead of writing a JIT that’s specific to the language you’re implementing, you generate one. Automatically.

RPython

- JIT compiler generator
- Useful primitives

In addition to the JIT generator. RPython has useful primitives for building the sort of things we need for a fast dynamic language.

Tracing JITs

So RPython generates a JIT for us. Specifically a tracing JIT. What is a tracing JIT? It's a JIT which observes the execution of a program (usually a loop at a time), and compiles linear code paths, with what are called "guards". What does that mean? Let's look at an example:

```
n = 10
while n != 1:
    if n & 1 == 0:
        n /= 2
    else:
        n = 3 * n + 1
```


So here we have a simple RPython loop which computes (sort of), the collatz conjecture. If you don't know that off hand, here's a loop with some math. This function is RPython, so these are all real machine ints, no dynamic type checking, or anything like this. Let's take a look at how this would get JIT'd

```python
n = 10
while n != 1:
    if n & 1 == 0:
        n /= 2
    else:
        n = 3 * n + 1
```

```
loop(n)

i0 = int_ne(n, 1)
guard_true(i0)

i1 = int_and(n, 1)
i2 = int_eq(i1, 0)
guard_true(i2)

i3 = int_div(n, 2)
jump(i3)
```


What are we looking at here, this sequence of instructions maps to one iteration of the loop on the left. So we check if n != 1, and we guard_true. What is a guard? The idea is that you map every "if" statement to a guard, and then when the guard fails you jump somewhere totally else. But usually this code just keeps plowing ahead.

Key insight:
~~Maybe~~
~~Probably~~
Almost certainly

Friday, September 20, 13

So, the key insight to efficient compilation of dynamic languages is that you need to be able to communicate to the compiler that a certain condition is ALMOST ALWAYS, but not actually always, true. There's no analog to this in most statically typed languages, this variable *always* has this type, this struct field is *always* in this condition. Dynamically typed languages are all about "probably".

```python
class Class(object):
    def __init__(self):
        self.methods = {}

    def add_method(self, name, m):
        self.methods[name] = m

    def find_method(self, name):
        return self.methods[name]

class Instance(object):
    def __init__(self, cls):
        self.cls = cls

    def send(self, name):
        return self.cls.find_method(name).call(self)
```


So here's our starting point for the ruby object model. We've got classes, and instance. Classes have a dict mapping names to methods, and send looks up a method on the class and calls it. This sucks, a dict lookup for every method call is slooooow, but 99.9% of the time with the same class and name we get the same result.

The primitives RPython gives us

So we want a way to express the “almost always” logic of find_method. We talked about guards in tracing JITs. Now we just need to bridge the gap, how do we express the issues of a dynamic language, in terms of these guards and other operations. To start we’ll look at what tools RPython gives us

`@jit.elidable`

Friday, September 20, 13

So the first hint we have is the ability to mark a function as elidable. Which is a word no one else uses. Basically a call to an elidable function must always be safe to be replaced with its result, or whats called referential transparency. An important thing to note however, is that it may still do things like caching.

```python
@jit.elidable
def find_method(self, name):
    return self.methods[name]
```


So the first thing we might try to do is something like this. Unfortunately this is wrong. We can redefine methods, so it's possible for two calls to find_method to have different results if you redefined the method in the middle. So we need more tools. It's also important to know that we can only replace calls if all the arguments are known to be constant. Right now neither self or name is known to be a constant.

```
jit.promote(x)
```


The next tool we have is called promotion. Basically this means you take the value you observed when running the program, and create a guard for it.

```python
def f(x):
    jit.promote(x)
    # serious computering here

f(10)
```

```
i0 = int_eq(x, 10)
guard_true(i0, x)
# computering goes here
```


So we define this f() function, and it promotes its argument, which generates this guard. What's the use of promotion? When something is very cheap to check, and usually the same. For example a given code path in aa dynamic language almost always has teh same type. This also pairs nicely with elidable.

```
_immutable_fields_ = ["field?"]
```


First, I want to apologize for the obvious ridiculousness of this syntax. And now I'll explain what the heck you're looking at. We call it: Quasi-immutable fields. Sounds super cool and confusing. So what's it do? The idea is sometimes you have a field which almost never changes. See that almost word again?

```
_immutable_fields_ = ["field?"]
```

Friday, September 20, 13

So RPython does a cool JIT. When you read the field in the JIT, it just deletes the read, replaces the read operation with the known value, and keeps track of the fact that it made this assumption. But when you *write* to the field, it invalidates any JIT code which contains this assumption.

Putting it all together

So those are the 3 hints. On top of which every optimization we do is built. The trick is they compose nicely. So what does an optimized method lookup look like?

```python
class Class(object):
    _immutable_fields_ = ["version?"]

    def __init__(self):
        self.methods = {}
        self.version = 0

    def add_method(self, name, m):
        self.methods[name] = m
        self.version += 1

    def find_method(self, name):
        return self._find_method(name, self.version)

    @jit.elidable
    def _find_method(self, name, version)
        return self.methods[name]

class Instance(object):
    def __init__(self, cls):
        self.cls = cls

    def send(self, name):
        cls = jit.promote(self.cls)
        return cls.find_method(name).call(self)
```


This is it. No joke. This is ALL the logic you need for method lookup to be basically free. So what did we change?

```
class Class(object):
    _immutable_fields_ = ["version?"]

    def __init__(self):
        self.methods = {}
        self.version = 0

    def add_method(self, name, m):
        self.methods[name] = m
        self.version += 1

    def find_method(self, name):
        return self._find_method(name, self.version)

    @jit.elidable
    def _find_method(self, name, version)
        return self.methods[name]

class Instance(object):
    def __init__(self, cls):
        self.cls = cls

    def send(self, name):
        cls = jit.promote(self.cls)
        name = jit.promote(name)
        return cls.find_method(name).call(self)
```

Friday, September 20, 13

We made about 6 lines of changes (they're in bold). So what did we do? We now have this version we update whenever we get a new method. We've made find_method elidable and it takes the version. And we promote an instances class before looking for a method. Let's take a step through calling these, and what the optimizer does.

```
my_object.a_method
```


So this is what we’re going to trace.

```python
def send(self, name):
    cls = jit.promote(self.cls)
    name = jit.promote(name)
    return cls.find_method(name).call(self)
```

```
p0 = getfield(my_obj, "cls")
i1 = ptr_eq(p0, Constant(MyClass))
guard_true(i1)
```

```python
def send(self, name):
    cls = jit.promote(self.cls)
    name = jit.promote(name)
    return cls.find_method(name).call(self)
```

```
p0 = getfield(my_obj, "cls")
i1 = ptr_eq(p0, Constant(MyClass))
guard_true(i1)
i2 = ptr_eq("a_method", Constant("a_method"))
guard_true(i2)
```

```
def find_method(self, name):
    return self._find_method(name, self.version)
```

```
p0 = getfield(my_obj, "cls")
i1 = ptr_eq(p0, Constant(MyClass))
guard_true(i1)
i2 = ptr_eq("a_method", Constant("a_method"))
guard_true(i2)
i3 = getfield(p0, "version")
call(_find_method, p0, i3)
```

```
p0 = getfield(my_obj, "cls")
i1 = ptr_eq(p0, Constant(MyClass))
guard_true(i1)
i2 = ptr_eq("a_method", Constant("a_method"))
guard_true(i2)
i3 = getfield(p0, "version")
call(_find_method, p0, i3)
```

```
p0 = getfield(my_obj, "cls")
i1 = ptr_eq(p0, Constant(MyClass))
guard_true(i1)
i2 = ptr_eq("a_method", Constant("a_method"))
guard_true(i2)
i3 = getfield(p0, "version")
call(_find_method, p0, i3)
```

```
p0 = getfield(my_obj, "cls")
i1 = ptr_eq(p0, Constant(MyClass))
guard_true(i1)
guard_true(True)
i3 = getfield(p0, "version")
call(_find_method, p0, i3)
```

```
p0 = getfield(my_obj, "cls")
i1 = ptr_eq(p0, Constant(MyClass))
guard_true(i1)
~~guard_true(True)~~
i3 = getfield(p0, "version")
call(_find_method, p0, i3)
```

```
p0 = getfield(my_obj, "cls")
i1 = ptr_eq(p0, Constant(MyClass))
guard_true(i1)
i3 = getfield(p0, "version")
call(_find_method, p0, i3)
```

```
p0 = getfield(my_obj, "cls")
i1 = ptr_eq(p0, Constant(MyClass))
guard_true(i1)
i3 = getfield(Constant(MyClass), "version")
call(_find_method, p0, i3)
```

```
p0 = getfield(my_obj, "cls")
i1 = ptr_eq(p0, Constant(MyClass))
guard_true(i1)
i3 = getfield(Constant(MyClass), "version")
call(_find_method, p0, i3)
```

```
p0 = getfield(my_obj, "cls")
i1 = ptr_eq(p0, Constant(MyClass))
guard_true(i1)
call(_find_method, Constant(MyClass), 10)
```

```
p0 = getfield(my_obj, "cls")
i1 = ptr_eq(p0, Constant(MyClass))
guard_true(i1)
call(_find_method, Constant(MyClass), 10)
```

3 instructions
25 lines of code

```
p0 = getfield(my_obj, "cls")
i1 = ptr_eq(p0, Constant(MyClass))
guard_true(i1)
```

The result?
topazruby.com

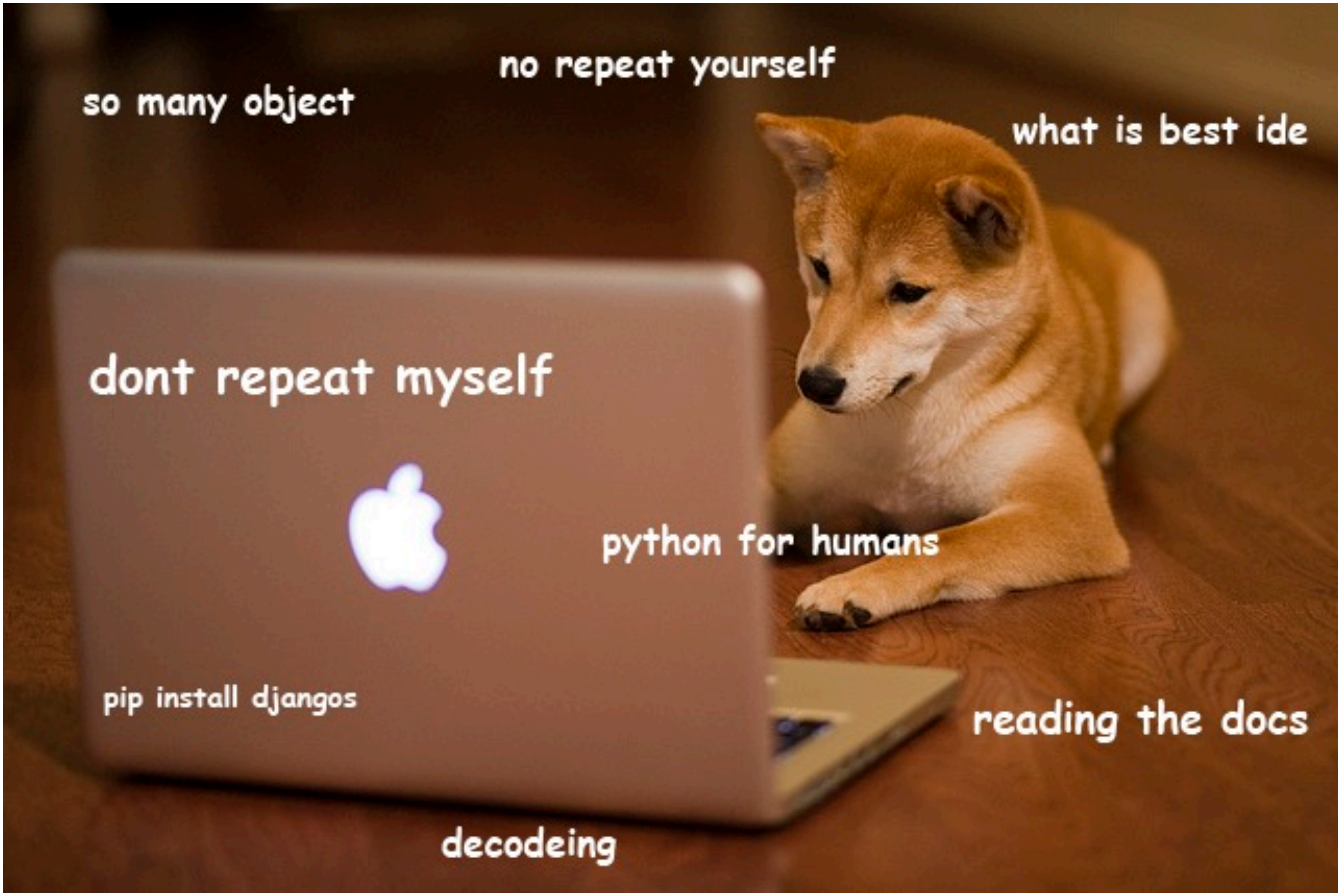
So the end result of all this work? A project I built called Topaz. It's a fast Ruby built on top of RPython. It's not complete, but I encourage you to check it out, contribute.

Other optimizations

- Fast CONSTANT lookups
- Fast, type-specialized, instance variable lookups
- Type-specialized containers

Miscellany

- pypy.org
- speed.pypy.org
- topazruby.com
- bitbucket.org/pypy/pypy
- github.com/topazproject/topaz



Thanks!

Friday, September 20, 13

Photo credit goes to Brian Curtin! Thanks for listening. Questions and answers now?