

# Using Types to Improve Performance with **Diamondback Ruby**

Seth Guenther  
CMSC 631

6 May 2009

# Ruby

- Interpreted
- Dynamically-typed
- Object-oriented (functional, imperative, reflective)
- Blend of Perl, Smalltalk, Ada, Eiffel, Lisp
- Created by Yukihiro “matz” Matsumoto in 1995
- Several implementations
  - Ruby – reference
  - JRuby
  - IronRuby

## Ruby (cont)

- Everything in Ruby is an object
  - No primitives
- Every operation is a method call
  - + : method on Fixnum object
- Dynamically-typed and Object-Oriented -> dynamic dispatch

# Dynamic Dispatch

- `x.foo()`
- Don't know type of `x` beforehand
- Multiple classes can implement `foo`
- Need to look at runtime type of `x` to determine which `foo` to call
  - Identify class of `x`
  - Find location of `foo` within definition of `x`
  - Follow super class chain if method not implemented by class of `x`
  - Metaprogramming facilities
- Every operation is method call = time consuming

# Dynamic Dispatch Optimizations

- Lookup caches
  - Maps (object type, method name) to method locations
  - Cache most recently used lookups
  - Method call
    - Look for entry in cache, use location if found
    - Do normal lookup routine otherwise; cache result

## Dynamic Dispatch Optimizations (cont)

- Better, but not as good as simple procedure call
  - Cache lookup
- Observation: locality of type usage
  - Method calls at some particular program point likely to be on objects of the same type
- Inline caches
  - Overwrite call to lookup routine with location of method
  - Add type checking code to beginning of method

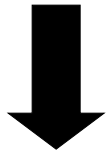
# Inline Caches

Interpreter

```
x.foo()  
call lookup routine
```

```
System Lookup Routine
```

cache



```
x.foo()  
call cached foo
```

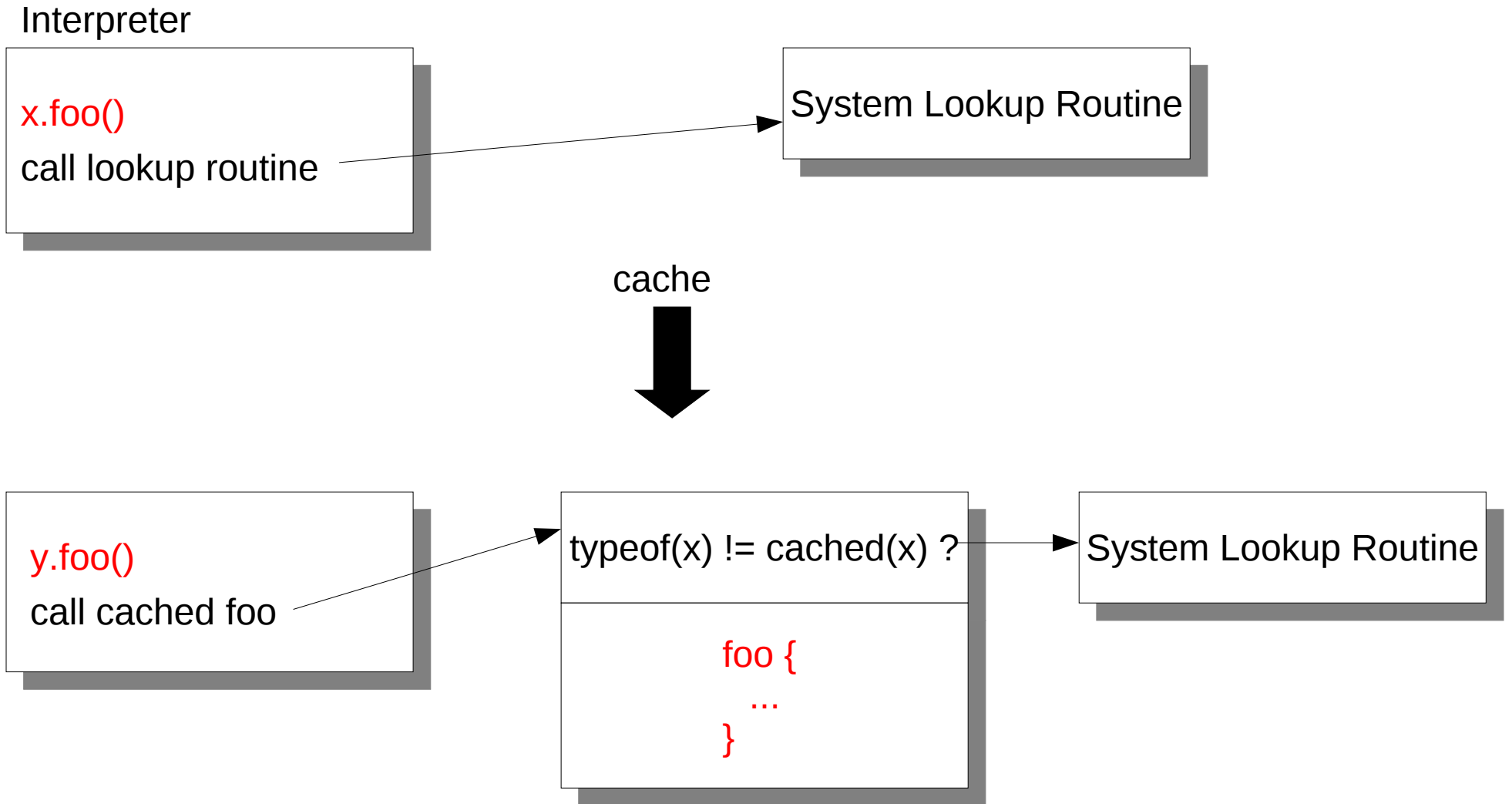
```
typeof(x) != cached(x) ?  
  
foo {  
  ...  
}
```

```
System Lookup Routine
```

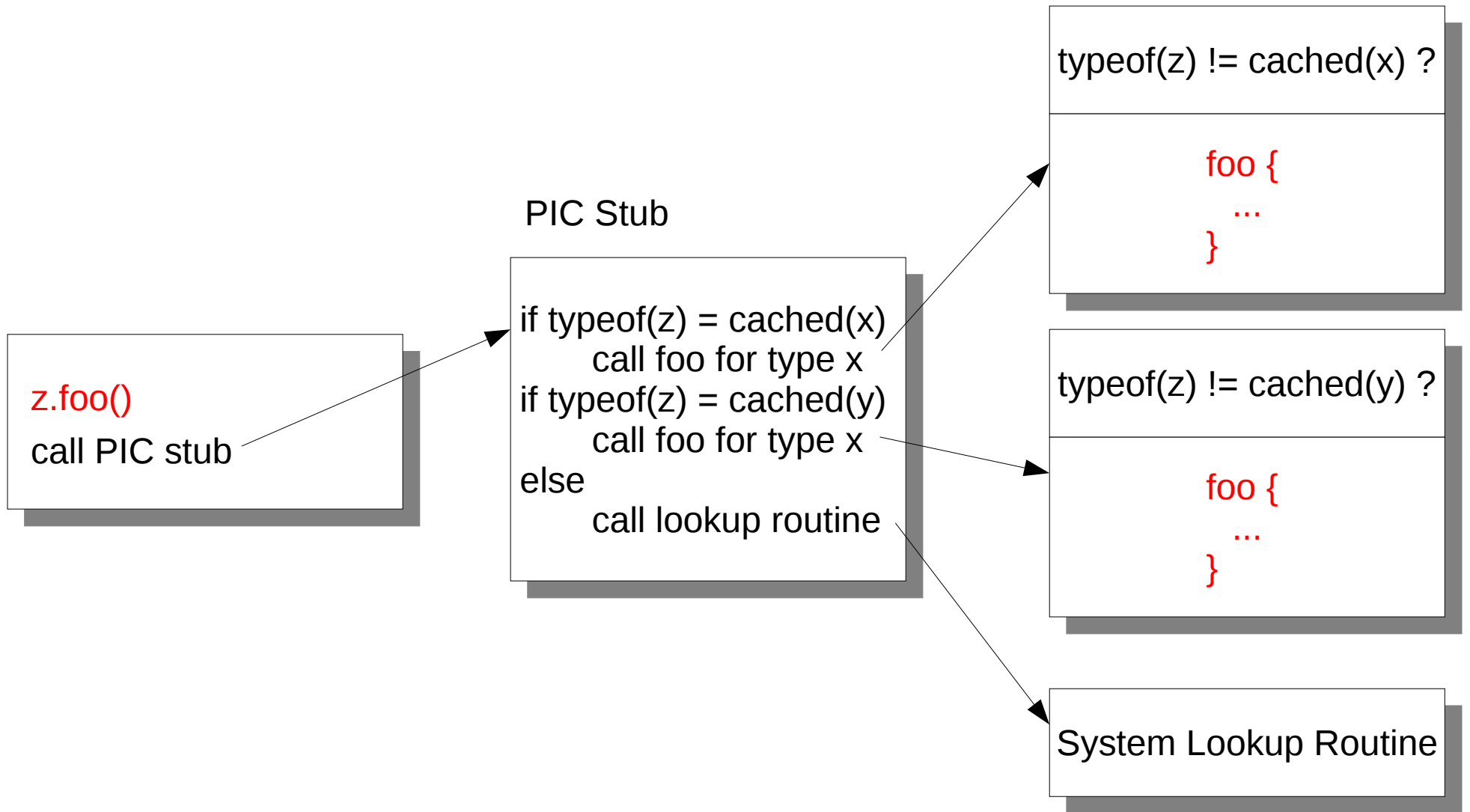
# Dynamic Dispatch Optimizations (cont)

- What if types are not relatively constant?
  - `foo = [ 1, false, 2 ]`  
`foo.each do |e| ... e.to_s ... end`
- Polymorphic call site
- Inline cache useless
  - Actually makes method call even slower due to useless cache lookup
- Polymorphic inline cache
  - Extend inline cache to hold multiple lookup results

# Polymorphic Inline Caches



# Polymorphic Inline Caches (cont)



## Polymorphic Inline Caches (cont)

- At each cache miss, add entry for new type to PIC stub
- No real need to evict from cache – number of types relatively small
  - Eventually cache will contain all types seen in program
-

# Project Idea 1

- Implement polymorphic inline caches in Ruby
  - Modify method lookup function to include PIC
  - Efficient cache lookup
- Problem
  - Metaprogramming
  - Add methods to/change methods of class during runtime

# Ruby Metaprogramming

```
class Duck
  def quack
    puts "quack!"
  end
end
```

```
d = Duck.new
d.quack
```

```
def d.bark
  puts "bark!"
end
```

```
d.bark
```

# Project idea 1

- Implement polymorphic inline caches in Ruby
  - Modify method resolution function to include PIC
  - Hash table
- Difficulty
  - Metaprogramming
  - Add methods to class during runtime
  - Would need to invalidate caches when class is modified

# Diamondback Ruby

- Extension to Ruby that adds static type system
  - Type inference
    - Constraint-based analysis
    - Inconsistent constraints represents type error
      - method not supported
      - wrong number of arguments to function
  - Type annotations (programmer specified)
    - “built-in” ruby types
- <http://www.cs.umd.edu/projects/PL/druby/>

## Project idea 2

- Use type information from Diamondback Ruby to augment process
- Main application – monomorphic call sites
  - Loops over monomorphic arrays
    - `foo.each do |e| ... end`
    - If every element of foo has the same type, don't even need cache

# Inline Caches

Interpreter

```
x.foo() @ loc  
call lookup routine
```

System Lookup Routine

cache

```
x.foo() @ loc  
call cached foo
```

```
x  
foo {  
  ...  
}
```

## Project idea 2

- Use type information from Diamondback Ruby to augment process
- Main application – monomorphic call sites
  - Loops over monomorphic arrays
    - `foo.each do |e| ... end`
    - If every element of `foo` has the same type, don't even need cache
- Mapping from program locations to “method resolution” function (function pointer)
  - Normally point to cache-enabled resolver
  - In above case, replace with jump to actual method

# Project Idea 3

- Benchmarks
  - Show improvement from caches
  - Show improvement from DRuby

# Progress so far...

- Learn Ruby
- Lots of reading
  - Papers (DRuby, Rubinius, PIC)
  - Ruby VM code
  - 
  - 
  -
- Made these slides
- Gave this progress report

# References

- U. Holzle, C Chambers and D. Ungar. Optimizing Dynamically-Typed Object-Oriented Languages with Polymorphic Inline Caches. Springer Verlag Lecture Notes in Computer Science 512, July 1991.
- M. Furr, J. An, J. Foster and M. Hicks. Static Type Inference for Ruby. ACM Symposium on Applied Computing, March 2009.
- <http://www.cs.umd.edu/projects/PL/druby/manual/manual.html>
- [http://en.wikipedia.org/wiki/Ruby\\_\(programming\\_language\)](http://en.wikipedia.org/wiki/Ruby_(programming_language))
- <http://www.hokstad.com/the-problem-with-compiling-ruby.html>
- <http://www.ruby-lang.org>

Questions?