Combining Static and Dynamic Typing in Ruby

Jeff Foster
University of Maryland, College Park

Joint work with Mike Furr, David An, Mike Hicks, Mark Daly, Avik Chaudhuri, and Ben Kirzhner

Introduction

- Scripting languages are extremely popular

	Lang	Rating		Lang	Rating
1	Java	17.3%	7	*Python	4.3%
2	C	16.6%	8	*Perl	3.6%
3	*PHP	10%	9	Delphi	2.7%
4	C++	9.5%	10	*JavaScript	2.6%
5	*Visual Basic	7.1%	11	*Ruby	2.4%
6	C#	5%	12	Objective-C	1.8%

*Scripting language TIOBE Index, January 2010 (based on search hits)

- Scripting languages are great for rapid development
  - Time from opening editor to successful run of the program is small
  - Rich libraries, flexible syntax, domain-specific support (e.g., regexps, syscalls)

Dynamic Typing

- Most scripting languages have *dynamic typing*
  - def foo(x)   y = x + 3; ...   # no decls of x or y
- Benefits
  - Programs are shorter

Java

```
class A {
  public static void main(String[] args) {
     System.out.println(“Hello, world!”);
} }
```

Ruby

```
puts “Hello, world!”
```

  - No type errors unless program about to “go wrong”
  - Possible coding patterns very flexible (e.g., eval(“x+y”))
  - Seems good for rapid development

Drawbacks

- Errors remain latent until run time

- No static types to serve as (rigorously checked) documentation

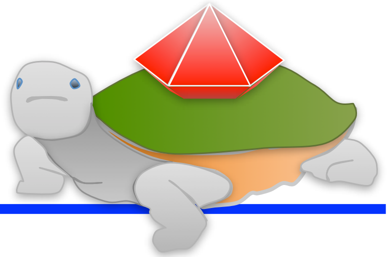
- Code evolution and maintenance may be harder
  - E.g., no static type system to find bugs in refactorings

- Performance can be significantly lower without sophisticated optimizations

Do these drawbacks matter?

- Getting an analysis correct is extremely important, particular when used for discovery
- Several highly public gaffes in recent years
  - Chang and collaborators **retracted 3 Science papers** and other articles **due to errors** in data analysis program (http://www.sciencemag.org/cgi/content/summary/314/5807/1856)
  - Commonly used family of substitution matrices for database searches and sequence alignments was **found to be incorrect 15 years after its introduction, due to software errors** in the tool that produced the data (http://www.nature.com/nbt/journal/v26/n3/full/nbt0308-274.html)
- Assurances that suggest a program is free of certain classes of errors would be most welcome

Diamondback Ruby (DRuby)



- Research goal: Develop a type system for scripting langs.
  - Simple for programmers to use
  - Flexible enough to handle common idioms
  - Provides useful checking where desired
  - Reverts to run time checks where needed
- DRuby: Statically checked and inferred types for Ruby
  - Ruby becoming popular, especially for building web apps
  - A model scripting language
    - Based on Smalltalk, and mostly makes sense internally
- RubyDust: DRuby types, but determined based on executions, not program analysis

This Talk

- Types for Ruby
  - Type system is rich enough to handle many common idioms
  - Relevant to other languages, e.g., Python and Javascript

- Inferring Ruby types
  - Static analysis plus profiling for dynamic feature characterization
  - Dynamic analysis for a more holistic, easier-to-deploy system

- Evaluation on a range of Ruby programs

Types for Ruby

- How do we build a type system that characterizes "reasonable" Ruby programs?
  - What idioms do Ruby programmers use?
  - Are Ruby programs even close to statically type safe?

- Goal: Keep the type system as simple as possible
  - Should be easy for programmer to understand
  - Should be predictable

Overview of the type system

- Standard stuff (think Java or C#): nominal types (i.e., class names), function and tuple types, generics
- Less standard:
  - Intersection and union types
  - Optional and vararg types
  - Structural object types
  - Types for mixins
  - Self type
  - Flow-sensitivity for local variables
- We'll illustrate our typing discipline on the core Ruby standard library

The Ruby Standard Library

- Ruby comes with a bunch of useful classes
  - Fixnum (integers), String, Array, etc.
- However, these are implemented in C, not Ruby
  - Type inference for Ruby isn't going to help!
- Our approach: type annotations
  - We will ultimately want these for regular code as well
- Standard annotation file base_types.rb
  - 185 classes, 17 modules, and 997 lines of type annotations

Basic Annotations

Type annotation

Block (higher-order method) type

```
class String
  ##% "+" : (String) → String

  ##% insert : (Fixnum, String) → String

  ##% upto : (String) {String → Object} → String
  ...
end
```

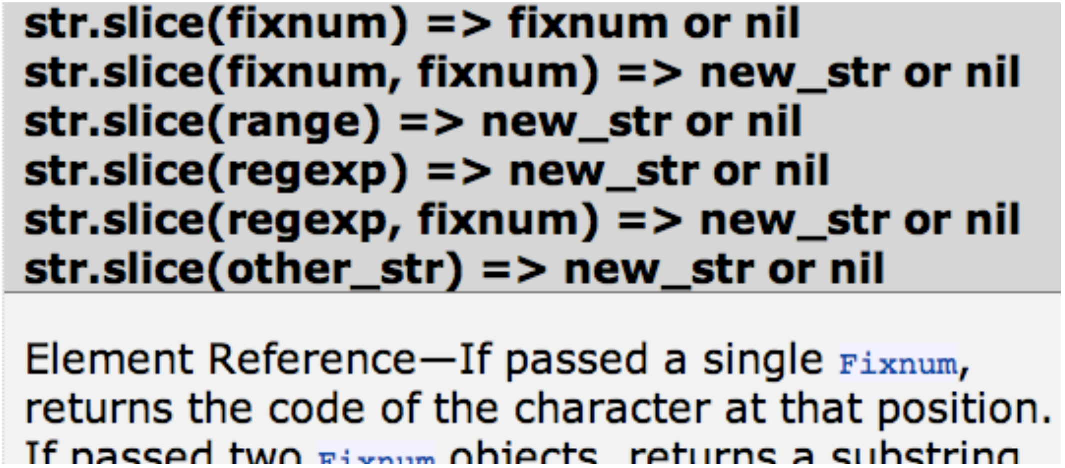
Intersection Types

```
class String
  include? : Fixnum → Boolean
  include? : String → Boolean
end
```

- Meth is *both* Fixnum → Boolean and String → Boolean
  - Ex: “foo”.include?(“f”); “foo”.include?(42);
- Generally, if x has type A and B, then
  - x is both an A and a B, i.e., x is a subtype of A and of B
  - and thus x has both A’s methods and B’s methods

Intersection Types (cont’d)

```
class String
  slice : (Fixnum) → Fixnum
  slice : (Range) → String
  slice : (Regexp) → String
  slice : (String) → String
  slice : (Fixnum, Fixnum) → String
  slice : (Regexp, Fixnum) → String
end
```



```
str.slice(fixnum) => fixnum or nil
str.slice(fixnum, fixnum) => new_str or nil
str.slice(range) => new_str or nil
str.slice(regexp) => new_str or nil
str.slice(regexp, fixnum) => new_str or nil
str.slice(other_str) => new_str or nil
```

Element Reference—If passed a single `Fixnum`, returns the code of the character at that position. If passed two `Fixnum` objects, returns a substring

- Intersection types are common in the standard library
  - 74 methods in base_types.rb use them
- Our types look much like the RDoc descriptions of methods
  - Except we type check the uses of functions
  - We found several places where the RDoc types are wrong
  - (Note: We treat nil as having any type)

Optional Arguments

```
class String
  chomp : () → String
  chomp : (String) → String
end
```

- Ex: “foo”.chomp(“o”);   “foo”.chomp();
  - By default, chomps $/

0 or 1 occurrence

- Abbreviation:

```
class String
  chomp : (?String) → String
end
```

Variable-length Arguments

```
class String
  delete : (String, *String) → String
end
```

0 or more occurrences

- Ex: “foo”.delete(“a”); “foo”.delete(“a”,“b”,“c”);
- *arg is equivalent to an unbounded intersection
- To be sensible
  - Required arguments go first
  - Then optional arguments
  - Then one varargs argument

Union Types

```
class A def f() end end
class B def f() end end
x = ( if ... then A.new else B.new)
x.f
```

- This method invocation is always safe
  - Note: in Java, would make interface I s.t. A < I, B < I
- Here x has type A or B
  - It's either an A or a B, and we're not sure which one
  - Therefore can only invoke x.m if m is common to both A and B
- Ex: Boolean short for TrueClass or FalseClass

Structural Subtyping

- Types so far have all been *nominal*
  - Refer directly to class names
  - Mostly because core standard library is magic
    - Looks inside of Fixnum, String, etc "objects" for their contents
- But Ruby really uses *structural* or *duck typing*
  - Basic Ruby op: method dispatch e0.m(e1, ..., en)
    - Look up m in e0, or in classes/modules e0 inherits from
    - If m has n arguments, invoke m; otherwise raise error
  - Most Ruby code therefore only needs objects with particular methods, rather than objects of a particular class

Object Types

```
module Kernel
  print : (*[to_s : () → String]) → NilClass
end
```

- print accepts 0 or more objects with a to_s method
- Object types are especially useful for native Ruby code:
  - def f(x)   y = x.foo;   z = x.bar;   end
  - What is the most precise type for f's x argument?
    - C1 or C2 or ... where Ci has foo and bar methods
      - Bad: closed-world assumption; inflexible; probably does not match programmer's intention
    - Fully precise object type: [foo:() →..., bar:()→...]

Diamondback Ruby

- Automatically infer the types of existing Ruby programs
  - Start with base_types.rb, then infer types for the rest of the code
- Implements *static type inference*
  - Analyze the source code and come up with types that capture *all* possible executions
  - Benefit: the types are sure to capture all behavior, even behavior not explicitly tested
  - Drawback: the technique is approximate, meaning that the system may fail to find types for correct programs

Dynamic Features

- We found that DRuby works well at the application level
  - Some experimental results coming up shortly

- But starts to break down if we analyze big libraries
  - Libraries include some interesting dynamic features
  - Typical Ruby program = small app + large libraries

Real-World Eval Example

```
class Format
  ATTRS = [“bold”, “underscore”,...]
  ATTRS.each do |attr|
    code = “def #{attr}() ... end”
    eval code
  end
end
```

Real-World Eval Example

```
class Format
  ATTRS = [“bold”, “underscore”,...]
  ATTRS.each do |attr|
    code = “def #{attr}() ... end”
    eval code
  end
end
```

```
class Format
  def bold() ... end
  def underline() end
end
```

Real-World Eval Example

```
class Format
  ATTRS = [“bold”, “underscore”,...]
  ATTRS.each do |attr|
    code = “def #{attr}() ... end”
    eval code
  end
end
```

- `eval` occurs at top level
- `code` can be arbitrarily complex
  - Thus we cannot generate a single static type for eval
- But, *in this case*, will always add the same methods
  - *Morally*, this *particular* code is static, rather than dynamic

Another Fun Example

```
config = File.read(__FILE__)
         .split(/__END__/).last
         .gsub(#\{(.*)\}/) { eval $1}
```

Another Fun Example

```
config = File.read(__FILE__)
         .split(/__END__/).last
         .gsub(#\{(.*)\}/) { eval $1}
```

Huh?

Another Fun Example

```
       config = File.read(__FILE__)
                .split(/__END__/).last
                .gsub(#\{(.*)\}/) { eval $1}
```

Read the current file

```
class RubyForge
  RUBYFORGE_D = File::join HOME, ".rubyforge"
  COOKIE_F    = File::join RUBYFORGE_D, "cookie.dat"
  config = ...
  ...
end
__END__
  cookie_jar : #{ COOKIE_F }
  is_private : false
  group_ids :
      codeforpeople.com : 1024
  ...
```

Another Fun Example

```
config = File.read(__FILE__)
         .split(/__END__/).last
         .gsub(#\{(.*)\}/) { eval $1}
```

```
class RubyForge
  RUBYFORGE_D = File::join HOME, ".rubyforge"
  COOKIE_F    = File::join RUBYFORGE_D, "cookie.dat"
  config = ...
  ...
end
__END__
  cookie_jar : #{ COOKIE_F }
  is_private : false
  group_ids :
      codeforpeople.com : 1024
  ...
```

Get everything after here

Another Fun Example

```
config = File.read(__FILE__)
         .split(/__END__/).last
         .gsub(#\{(.*)\}/) { eval $1}
```

```
class RubyForge
  RUBYFORGE_D = File::join HOME, ".rubyforge"
  COOKIE_F    = File::join RUBYFORGE_D, "cookie.dat"
  config = ...
  ...
end
__END__
  cookie_jar : #{ COOKIE_F }
  is_private : false
  group_ids :
      codeforpeople.com : 1024
  ...
```

Substitute this

Another Fun Example

```
config = File.read(__FILE__)
         .split(/__END__/).last
         .gsub(#\{(.*)\}/) { eval $1}
```

```
class RubyForge
  RUBYFORGE_D = File::join HOME, ".rubyforge"
  COOKIE_F    = File::join RUBYFORGE_D, "cookie.dat"
  config = ...
  ...
end
__END__
  cookie_jar : #{ COOKIE_F }
  is_private : false
  group_ids :
      codeforpeople.com : 1024
  ...
```

With this

Another Fun Example

```
config = File.read(__FILE__)
             .split(/__END__/).last
             .gsub(#\{(.*)\}/) { eval $1}
```

```
class RubyForge
  RUBYFORGE_D = File::join HOME, ".rubyforge"
  COOKIE_F    = File::join RUBYFORGE_D, "cookie.dat"
  config = ...
  ...
end
__END__
  cookie_jar : “/home/jfoster/.rubyforge/cookie.dat”
  is_private : false
  group_ids :
      codeforpeople.com : 1024
  ...
```

Eval it

Another Fun Example

```
config = File.read(__FILE__)
         .split(/__END__/).last
         .gsub(#\{(.*)\}/) { eval $1}
```

```
class RubyForge
  RUBYFORGE_D = File::join HOME, ".rubyforge"
  COOKIE_F    = File::join RUBYFORGE_D, "cookie.dat"
  config = ...
  ...
end
__END__
  cookie_jar : “/home/jfoster/.rubyforge/cookie.dat”
  is_private : false
  group_ids :
      codeforpeople.com : 1024
  ...
```

Store in config

Profiling Dynamic Features

- To handle eval and similar features, we extend DRuby static inference to incorporate profiling information
  - When eval(...) occurrences are reached, we replace them with the code the evaluated to during test runs, and perform inference on that code
- Found that in most situations, eval was not unconstrained, but idiomatic. In short, the technique worked well

Example Errors Found

- Typos in names
  - Archive::Tar::ClosedStream instead of Archive::Tar::MiniTar::ClosedStream
  - Policy instead of Policies

- Other standard type errors

```
return rule_not_found if !@values.include?(value)
```

  - rule_not_found not in scope
  - Program did include a test suite, but this path not taken

Syntactic Confusion

```
assert_nothing_raised { @hash[‘a’, ‘b’] = 3, 4 }
...
assert_kind_of(Fixnum, @hash[‘a’, ‘b’] = 3, 4)
```

- First passes [3,4] to the []= method of @hash
- Second passes 3 to the []= method, passes 4 as last argument of assert_kind_of
  - Even worse, this error is suppressed at run time due to an undocumented coercion in assert_kind_of

Syntactic Confusion (cont’d)

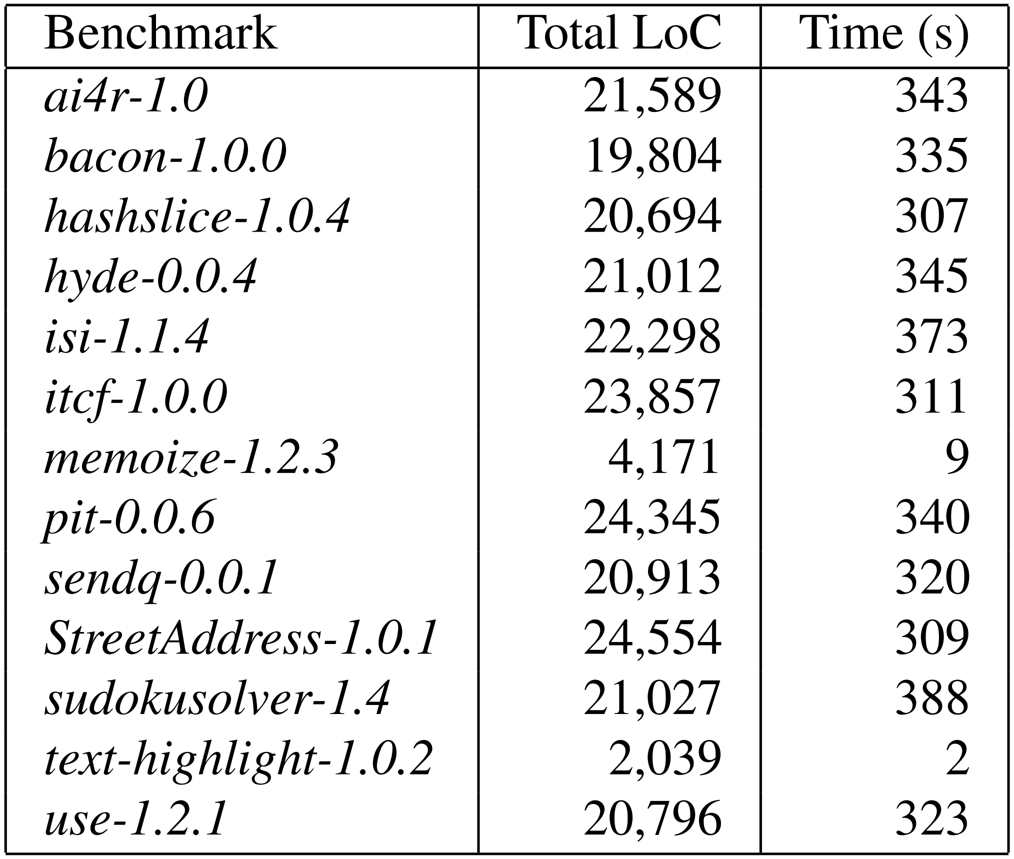
```
flash[:notice] = “You do not have ...“
+ “...”
```

- Programmer intended to concatenate two strings
- But here the + is parsed as a unary operator whose result is discarded

```
@count, @next, @last = 1
```

- Intention was to assign 1 to all three fields
- But this actually assigns 1 to @count, and nil to @next and @last

Performance (DRuby)



Benchmark	Total LoC	Time (s)
ai4r-1.0	21,589	343
bacon-1.0.0	19,804	335
hashslice-1.0.4	20,694	307
hyde-0.0.4	21,012	345
isi-1.1.4	22,298	373
itcf-1.0.0	23,857	311
memoize-1.2.3	4,171	9
pit-0.0.6	24,345	340
sendq-0.0.1	20,913	320
StreetAddress-1.0.1	24,554	309
sudokusolver-1.4	21,027	388
text-highlight-1.0.2	2,039	2
use-1.2.1	20,796	323

- Times include analysis of all standard library code used by app

Follow-on Work

- DRails — Type inference for Ruby on Rails
  - Rails is a popular web application framework
- User study — Is type inference useful?
  - The jury is still out
- Rubydust — Static type inference, at run time
  - Ruby *library* that does type inference, rather than a separate tool
- Rubyx — Symbolic execution for Ruby
  - Powerful technology that extends testing
  - Used to find security vulnerabilities in Rails programs
  - But can be used for many program reasoning tasks

http://www.cs.umd.edu/projects/PL/druby