

Adding Static Typing to Ruby

Jeff Foster
University of Maryland, College Park

Joint work with Mike Furr, David An, and Mike Hicks

Introduction

- Scripting languages are extremely popular

	Lang	Rating		Lang	Rating
1	Java	19.0%	7	*Python	4.7%
2	C	15.9%	8	*Perl	4.3%
3	C++	10.1%	9	*JavaScript	3.4%
4	*Visual Basic	9.2%	10	Delphi	3.3%
5	*PHP	8.9%	11	*Ruby	3.1%
6	C#	5.6%	12	D	1.0%

*Scripting language

TIOBE Index, January 2009 (based on search hits)

- Scripting languages are great for rapid development
 - 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

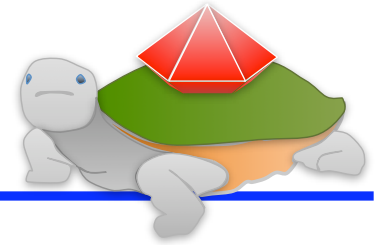
```
puts "Hello, world!"
```

- No type errors unless program about to “go wrong”
 - Possible coding patterns very flexible
 - Good for rapid development

Dynamic Typing (cont'd)

- Drawbacks
 - Errors remain latent until run time
 - No static types to serve as documentation
 - Code maintenance may be harder
 - E.g., no static type system to ensure refactorings are type correct

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: Adding static types to Ruby
 - Ruby becoming popular, especially for building web apps
 - A model scripting language
 - Based on Smalltalk, and mostly makes sense internally

This Talk

- RIL: The Ruby Intermediate Language
 - Small, easy to analyze subset of Ruby
- Static type inference for Ruby [OOPS 2009]
 - Type system is rich enough to handle many common idioms
 - Ruby apps are mostly statically typable with DRuby
- Profile-based analysis for highly dynamic features [Unpub]
 - Ruby includes reflection, eval(), many hard-to-analyze features
 - Hypothesis: Features are not as dynamic as they seem
 - Use profiles to gather data from test runs, then analyze statically

Ruby Intermediate Language (RIL)

- “[Ruby should] feel natural to programmers” — Yukihiro Matsumoto
 - Result: Grammar not amenable to LL/LR parsing
 - Ruby’s own parser is complex, written in C, tied to interpreter
- Solution: A GLR parser for Ruby
 - Grammar productions may be ambiguous
 - Ambiguities resolved eventually to yield one final parse tree
- Parser statistics
 - 63 productions, 411 other LoC
 - 317 hand-written tests to ensure ASTs correct
 - Ran on 1,239 Ruby files (163,297 LoC)
 - Had to manually disambiguate 12 locations

Ruby Intermediate Language (RIL)

- Ruby has many ways to do the same thing
 - `if p then e / e if p / unless (not p) e / e unless (not p)`
- Control flow in Ruby can be complex
 - In `w = x().y(z())` does `x()` or `z()` occur first?
 - Need to know this to build flow-sensitive analyses
- Ruby has some weird behavior
 - `x = a` # error if `a` undefined
 - `if false then a = 3 end; x = a;` # sets `x` to `nil` (!)
- RIL: Simplifies this all away
 - 24 stmt kinds, each with only one side effect, organized as CFG
 - **Much easier to analyze than unsimplified Ruby**

Static Types for Ruby

- How do we build a static type system that accepts “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
- 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)

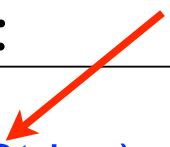
Optional Arguments

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

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

- Abbreviation: 0 or 1 occurrence

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



Aside: \$ in Ruby

- Global variables begin with \$
- Here are all the special global variables formed from non-ascii names
 - \$! @\$; \$, \$/ \$\ \$. \$_ \$< \$> \$\$
 - \$? \$~ \$= \$* `\$' \$+ \$& \$0 \$: \$"
 - \$1 \$2 \$3 \$4 \$5 \$6 \$7 \$8 \$9 (these are local)

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:() → ...]`

The Self Type

```
module Kernel
  clone : () → self
end
```

- Ex: `class A ... end ... x = A.new.clone`
 - `x` should have type `A`, not `Object` or `Kernel`
 - The `self` type makes this happen
- Implemented internally with parametric polymorphism
 - `clone : ∀u . (self : u) → u`

Parametric Polymorphism (Generics)

Type parameter

```
class Array<t>
  at : (Fixnum) → t

  first : () → t
  first : (Fixnum) → Array<t>

  collect<u> : () { t → u } → Array<u>
  "+"<u> : (Array<u>) → Array(<t or u>)
end
```

Type
instantiation

Method
polymorphism

Tuple Types

```
def f() [ I, true ] end  
a, b = f                # a = I, b = true
```

- $f : () \rightarrow \text{Array}\langle \text{Fixnum or Boolean} \rangle$?
 - Not precise enough to type above example
- $f : () \rightarrow \text{Tuple}\langle \text{Fixnum, Boolean} \rangle$
 - $\text{Tuple}\langle t_1, \dots, t_n \rangle =$ array where elt i has type t_i
- Implicit subtyping between **Tuple** and **Array**
 - $\text{Tuple}\langle t_1, \dots, t_n \rangle < \text{Array}\langle t_1 \text{ or } \dots \text{ or } t_n \rangle$

That's the Basic Type System

- Optional and varargs
- Intersection and union types
- Object types
- The self type
- Parametric polymorphism (generics)
- Tuple types
- (Plus types for mixins, first-class method types, flow-sensitivity for local variables)
- A fair amount of machinery, but not too bad!

Dynamic Features

- The basic type system 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

Require

- Ruby programs load files by calling `require`
 - `require "foo.rb" # loads file foo.rb`
- `Require` is actually just a special method
 - So it can be nearly impossible to statically determine loaded files

```
require File.join(File.dirname(__FILE__), '..', 'lib', 'sudokusolver')
```

```
Dir.chdir("../") if base == "test"  
$LOAD_PATH.unshift(Dir.pwd + "/lib"); ...;  
require "memoize"
```

- First `require` call loads file named by dynamically computed string
- Second `require` loads file from new location because path was side-effected
 - Tricks like this are heavily used by the *rubygems* package manager

Send and Eval

- Calling `send` performs reflective method invocation

```
def initialize(args)
  args.keys.each do |attrib|
    self.send("#{attrib}=", args[attrib])
  end
end
```

- Ruby also lets user `eval` strings to execute code

```
ATTRIBUTES = ["bold", "underscore", ...]
ATTRIBUTES.each do |attr|
  code = "def #{attr}(&blk) ... end"
  eval code
end
```

- Notice: dynamic code affects safety of static code
- Cannot be handled with, e.g., gradual typing

Method_missing

- If `method_missing` is defined, intercepts calls that go to undefined methods

```
def method_missing(mid, *args)
  mname = mid.id2name
  if mname =~ /=$/
    ...
    @table[mname.chop!.intern] = args[0]
  elsif args.length == 0
    @table[mid]
  else
    raise NoMethodError, "undefined method.."
  end
end
```

The Problem

- These language constructs are hard to analyze statically
 - `Require`, `send`, `eval` would need some kind of static string tracking
 - Several proposals in the literature based on regular expressions
 - Seems unlikely we'd get precise enough information out
 - Would need to modify type inference to process imprecise ASTs
 - Could treat `method_missing` as intercepting all unhandled calls
 - But we might suppress errors for calls that really aren't handled

Profile-Guided Typing

- Idea: These constructs are not as dynamic as they seem
 - If we fix the Ruby installation and particular client application...
 - ...then maybe the dynamic behavior becomes fixed
- Use profiling to gather profile of strings for dyn. constrs.
 - Run program under set of *test cases* supplied by the user
 - After all, tests are a kind of specification programmer already writes
 - Seems fair to ask for tests to exhibit dynamic behavior, plus easy to understand
 - Perform type inference using strings from profile
 - We actually transform the program and then parse it
 - E.g., replace `eval s` by the contents of `s`
 - Run program with checks to enforce conformance to profile

Results, Part I: Type Inference Only

- Applied DRuby to set of small benchmarks
 - From RubyForge and colleagues
 - Use stub file to model standard library
 - Analysis carried out before without profiling support, which is really needed for the standard library
- Needed to make a few changes to benchmarks:
 - Add stub file to load multiple files or simulate unit tests
 - No profiling yet, so
 - Manually expand metaprogramming code
 - Replace `require x` with `require "string"`

Results, Part I (cont'd)

Program	LOC	Changes	Tm(s)	E	W	FP
<i>pscan-0.0.2</i>	29	None	3.7	0	0	0
<i>hashslice-1.0.4</i>	91	S	2.2	1	0	2
<i>sendq-0.0.1</i>	95	S	1.9	0	3	0
<i>merge-bibtex</i>	103	None	2.6	0	0	0
<i>substitution_solver-0.5.1</i>	132	S	2.5	0	4	0
<i>style-check-0.11</i>	150	None	2.7	0	0	0
<i>ObjectGraph-1.0.1</i>	153	None	2.3	1	0	1
<i>relative-1.0.2</i>	158	S	2.3	0	1	5
<i>vimrecover-1.0.0</i>	173	None	2.8	2	0	0
<i>itcf-1.0.0</i>	183	S	4.7	0	0	1
<i>sudokusolver-1.4</i>	201	R-1, S	2.7	0	1	1
<i>rawk-1.2</i>	226	None	3.1	0	0	2
<i>pit-0.0.6</i>	281	R-2, S	5.3	0	0	1
<i>rphotoalbum-0.4</i>	313	None	12.6	0	1	0
<i>gs_phone-0.0.4</i>	827	S	37.2	0	0	0
<i>StreetAddress-1.0.1</i>	877	R-1, S	6.4	0	0	0
<i>ai4r-1.0</i>	992	R-10, S	12.2	1	6	1
<i>text-highlight-1.0.2</i>	1,030	M-2, S	14.0	0	0	2

M—manually expanded meta-programming code

R—replaced require argument with constant String

S—added stub file to trigger multiple files and/or simulate unit test

Errors — 5 Total

- 3 due to undefined variables

- Example from *ai4r*

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

- `rule_not_found` not in scope
- Program does include test suite, but did not take this path
- Errors in *vimrecover* in error handling code
 - Actual error suppressed by undefined variable exception!

Errors — 5 Total (cont'd)

- | due to 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`

Errors — 5 Total (cont'd)

- I due to odd loop exit

```
$baseClass = ObjectSpace.each_object(Class)
{ |k| break k if k.name == baseClassName }
```

- Code block intended to terminate via `break k`
 - Returns an instance of `Class`
- But code block could terminate normally
 - Then `each_object` returns a `Fixnum` (number of elements visited)

Warnings — 16 total

- 14 due to missing code block arguments

```
5.times { |i| print "*" }
```

- Above code specifies argument, but does not use it
- Many Ruby programs omit the argument completely in this case:

```
5.times { print "*" }
```

- If we allow this, cannot find bugs where code block called with wrong number of arguments
 - We feel this is bad style, since it is very easy to fix
- 1 case where Ruby allows code we consider confusing
 - 1 case where loop always exited by break

False Positives — 16 total

- 3 due to union types resolved dynamically
 - Methods return either false or an array
 - Clients check return value against false before using
- 3 due to method redefinitions
 - Currently forbidden in DRuby; it cannot decide which method is actually called
- Various causes of remaining false positives
 - Could not resolve use of intersection type
 - Could not locate definition of a constant
 - Wrapper around `require` that dynamically changed argument
 - Method `new` rebound

Results, Part 2 (Profiling): Occurrences

- Used previous benchmarks plus some new programs with test suites
 - Need dynamic runs to gather profiling data

Benchmark	(Syntactic Occ/Unique Strings)					Lib
	LoC	Req	Eval	Snd	Tot	Tot
<i>ai4r-1.0</i>	748	3/ 3	2/ 2	4/ 4	9/ 9	31/361
<i>bacon-1.0.0</i>	258	0/ 0	0/ 0	0/ 0	0/ 0	29/339
<i>hashslice-1.0.4</i>	78	0/ 0	0/ 0	0/ 0	0/ 0	30/347
<i>hyde-0.0.4</i>	144	2/ 2	1/11	1/ 2	4/15	31/343
<i>isi-1.1.4</i>	224	0/ 0	1/ 1	0/ 0	1/ 1	31/340
<i>itcf-1.0.0</i>	178	0/ 0	0/ 0	0/ 0	0/ 0	38/495
<i>memoize-1.2.3</i>	69	0/ 0	0/ 0	0/ 0	0/ 0	12/ 16
<i>pit-0.0.6</i>	166	2/ 2	0/ 0	0/ 0	2/ 2	34/494
<i>sendq-0.0.1</i>	90	0/ 0	0/ 0	0/ 0	0/ 0	31/343
<i>StreetAddress-1.0.1</i>	875	1/ 1	0/ 0	1/15	2/16	38/493
<i>sudokusolver-1.4</i>	188	2/ 2	1/ 1	0/ 0	3/ 3	31/344
<i>text-highlight-1.0.2</i>	249	0/ 0	2/48	0/ 0	2/48	0/ 0
<i>use-1.2.1</i>	193	0/ 0	0/ 0	0/ 0	0/ 0	30/350
Total	3,460	10/10	7/63	6/21	23/94	41/607

Coverage of Dynamic Behavior

Benchmark	Uniq strs	Load tst cov	Tsts for full cov
<i>ai4r-1.0</i>	352	99%	9 of 18
<i>bacon-1.0.0</i>	339	100%	–
<i>hashslice-1.0.4</i>	338	100%	–
<i>hyde-0.0.4</i>	355	99%	1 of 3
<i>isi-1.1.4</i>	340	100%	–
<i>itcf-1.0.0</i>	492	100%	–
<i>memoize-1.2.3</i>	12	100%	–
<i>pit-0.0.6</i>	492	100%	–
<i>sendq-0.0.1</i>	339	100%	–
<i>StreetAddress-1.0.1</i>	508	97%	1 of 1
<i>sudokusolver-1.4</i>	342	100%	–
<i>text-highlight-1.0.2</i>	48	100%	–
<i>use-1.2.1</i>	338	100%	–

- Load test = load in source file, but don't run test
 - On 10/13 benchmarks, covers all behavior seen in test suite
 - For other benchmarks, does not require that many tests
- Caveat: *Except* **send** calls to test suite methods

Type Inference with Profiling

Benchmark	LoC	Time (s)	Errors
<i>ai4r-1.0</i>	16,011	152	200
<i>bacon-1.0.0</i>	14,265	144	195
<i>hashslice-1.0.4</i>	15,131	145	196
<i>hyde-0.0.4</i>	15,449	194	196
<i>isi-1.1.4</i>	16,773	195	207
<i>itcf-1.0.0</i>	18,395	123	213
<i>memoize-1.2.3</i>	3,938	29	20
<i>pit-0.0.6</i>	16,785	223	202
<i>sendq-0.0.1</i>	15,353	164	202
<i>StreetAddress-1.0.1</i>	19,092	117	214
<i>sudokusolver-1.4</i>	15,465	203	201
<i>text-highlight-1.0.2</i>	1,922	3	1
<i>use-1.2.1</i>	15,232	146	195

- Can analyze much more code, errors are false positives
 - Working on shaking these out
 - Common idiom: Code tests `RUBY_VERSION`, then has two (type inconsistent) behaviors depending on the version

Some Future Directions

- Enhancing DRuby to deal with new set of false positives
 - Idea: Only analyze code seen in dynamic runs
 - Would eliminate errors due to Ruby version
 - Idea: Enhance type system, perhaps with things like occurrence types, to handle various idioms
- Apply to Ruby on Rails
 - Popular web app framework, relies heavily on dynamic features
- User studies
 - Is DRuby actually helpful for developers?

Conclusion

- DRuby: Static and dynamic typing for Ruby
 - Built clean and robust GLR parser for Ruby
 - Transform Ruby into RIL to simplify analyses
 - Developed type grammar for core std lib
 - Perform type inference
 - Profiling to handle highly dynamic features
- Promising initial results
 - Ran inference on 18 applications [OOPS 2009]
 - All are completely or nearly statically typable
 - Examined profiles for 13 applications [Unpublished]
 - Most of the time, only need one exemplar run to see all dynamic behavior

For More Information

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

- Links to papers, etc
- Implementation available in March
 - Let us know if you find any or all of it useful!