Combining Static and Dynamic Typing in Ruby: Two Approaches

Jeff Foster
University of Maryland, College Park
Introduction

- Scripting languages are extremely popular
  
<table>
<thead>
<tr>
<th>Lang</th>
<th>Rating</th>
<th>Lang</th>
<th>Rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>17%</td>
<td>*Python</td>
<td>3.2%</td>
</tr>
<tr>
<td>Java</td>
<td>16.2%</td>
<td>*JavaScript</td>
<td>2%</td>
</tr>
<tr>
<td>C++</td>
<td>8.7%</td>
<td>Transact-SQL</td>
<td>2%</td>
</tr>
<tr>
<td>Objective-C</td>
<td>8.6%</td>
<td>*VB .NET</td>
<td>1.8%</td>
</tr>
<tr>
<td>*PHP</td>
<td>6.4%</td>
<td>*Perl</td>
<td>1.7%</td>
</tr>
<tr>
<td>C#</td>
<td>5.6%</td>
<td>*Ruby</td>
<td>1.4%</td>
</tr>
<tr>
<td>*Visual Basic</td>
<td>4.8%</td>
<td>Object Pascal</td>
<td>0.9%</td>
</tr>
</tbody>
</table>

*Scripting language  

- 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*
  
  ```python
  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
  
  - 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 ensure refactorings are type correct
Diamondback Ruby
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
First Part of This Talk

• RIL: The Ruby Intermediate Language
  ▪ Small, easy to analyze subset of Ruby

• Static type inference for Ruby
  ▪ Type system is rich enough to handle many common idioms

• Profile-based analysis for highly dynamic features
  ▪ Reflection, eval, method_missing, etc

• Joint work with Mike Furr, David An, Mike Hicks, Mark Daly, Avik Chaudhuri, and Ben Kirzhner
On The Design of Ruby

• In usual language design, there are several good properties:

<table>
<thead>
<tr>
<th>consistency</th>
<th>simplicity</th>
</tr>
</thead>
<tbody>
<tr>
<td>orthogonality</td>
<td>flexibility</td>
</tr>
<tr>
<td>succinctness</td>
<td>intuitiveness</td>
</tr>
<tr>
<td>DRY (Don’t Repeat Yourself)</td>
<td>good names</td>
</tr>
<tr>
<td>generalness</td>
<td>naturalness</td>
</tr>
</tbody>
</table>

meets programmers’ common sense

In the design policy of Ruby, they are also good properties.

—Akira Tanaka, Language and Library API Design for Usability of Ruby, PLATEAU 2009
On The Design of Ruby (cont’d)

• However, sometimes Ruby overrides the properties for usability.
  ▪ Ruby [doesn’t] need consistency including rare usage.
  ▪ Ruby [doesn’t] need succinctness including rare usage.
  ▪ Ruby [doesn’t] need orthogonality including rare usage.
  ▪ Ruby [doesn’t] need simplicity including rare usage.
Ruby Intermediate Language (RIL)

• A front-end for Ruby code analysis and transformation

• Key features
  ▪ GLR parser for Ruby source code
  ▪ Compact, simplified intermediate representation
  ▪ Pretty-printer that outputs valid, executable Ruby code
  ▪ Partial reparsing module to make code transformation easier
  ▪ Dataflow analysis engine
  ▪ Support for run-time profiling
Parsing Ruby Source

- “[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
Intermediate Representation

• 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

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

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

  ###% upto : (String) {String → Object} → String

  ...
end
Intersection Types

- 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

```ruby
class String
  include? : Fixnum → Boolean
  include? : String → Boolean
end
```
Intersection Types (cont’d)

- 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

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

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

- Abbreviation:

```ruby
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

- 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

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

0 or more occurrences
Union Types

- 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

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

- `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:() →...]`
Tuple Types

```ruby
def f() [ 1, true ] end
a, b = f              # a = 1, b = true
```

- \( f : () \rightarrow \text{Array}<\text{Fixnum or Boolean}> \)?
  - Not precise enough to type above example
- \( f : () \rightarrow \text{Tuple}<\text{Fixnum}, \text{Boolean}> \)
  - \( \text{Tuple}<t_1, ..., t_n> = \text{array where elt } i \text{ has type } t_i \)
- **Implicit subtyping between Tuple and Array**
  - \( \text{Tuple}<t_1, ..., t_n> < \text{Array}<t_1 \text{ or } ..., \text{ or } t_n> \)
That’s the Basic Type System

- Optional and varargs
- Intersection and union types
- Object types
- Tuple types
- (Plus the self type, parametric polymorphism (generics), 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
class Format
  ATTRS = ["bold", "underscore", ...]
  ATTRS.each do |attr|
    code = "def #{attr}() ... end"
    eval code
  end
end
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
• But, will always add the same methods
  ▪ *Morally*, this code is static, rather than dynamic
• Idea: execute the code and see what **eval** does
  ▪ Augment static analysis with this information
Profile-Guided Static Analysis

class Format
  ATTRS = ["bold", "underscore", ...]
  ATTRS.each do |attr|
    code = "def #{attr}() ... end"
    if code = "def bold() ... end"
      def bold() ... end
    else if code = "def underscore()..."
      def underscore() ... end
    else
      safe_eval code
    end
  end
end
else case adds extra dynamic checks

```ruby
class Format
  ATTRS = ["bold", "underscore", ...]
  ATTRS.each do |attr|
    code = "def #{attr}() ... end"
    if code = "def bold() ... end"
      def bold() ... end
    else if code = "def underscore()..."
      def underscore() ... end
    else
      safe_eval code
      end
  end
end
```

Checks ensure that any runtime type error *blames* a string passed to `safe_eval`
Theory of Profiling System

- **Theorem:** Translation is *faithful*
  - Static analysis is seeing a correct projection of the actual runtime behavior

- **Theorem:** Translation + static typing is sound
  - Program either executes without getting stuck, or can blame string that wasn’t seen before
Profiling Effectiveness

- Analyzed 13 benchmarks, including std-lib code
  - 24,895 LOC in total
  - Found 66 uses of dynamic features

- Inspected each dynamic feature and categorized how “dynamic” the usage really was
  - Found all uses could be divided into a few categories
  - All of which are morally static
Performance

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

• LoC = application + libraries it uses

• Type inference problem actually quite complicated
  • These times are reasonable for whole-program analysis
Example Errors Found

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

• Other standard type errors
  
  ```ruby
  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
Example Errors Found (cont’d)

- Model **Post** does not exist in the Rails app

```ruby
class Integer
  def to_bn
    OpenSSL::BN.new(self)
  end
end
```

- **BN.new** expects **String**, not **Integer**
- **3.to_bn** would cause a type error
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)

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

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

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

```ruby
@count, @next, @last = 1
```
DRuby Was Promising, But...

• Ruby front-end somewhat fragile
  - Tied to Ruby 1.8.7; required fixing for Ruby 1.9, would need to fix again for Ruby 2.0

• Phased analysis doesn’t match that well with many uses of Ruby
  - Ruby has no “compile” phase, so somewhat unnatural

• Limitations of type system require adding a lot more sophistication and complexity
  - Complex type system = hard for programmer to predict