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

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

*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         | Ruby              |
    |--------------|-------------------|
    | `class A {`  | `puts “Hello, world!”` |
    |  `public static void main(String[] args) {` |         |
    |  `  System.out.println(“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

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

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

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

Type annotation

Block (higher-order method) type
Intersection Types

• Meth is both \texttt{Fixnum} $\rightarrow$ \texttt{Boolean} and \texttt{String} $\rightarrow$ \texttt{Boolean}
  
  \begin{itemize}
  \item Ex: “foo”.include?(“f”); “foo”.include?(42);
  \end{itemize}

• Generally, if $x$ has type $A$ and $B$, then
  
  \begin{itemize}
  \item $x$ is both an $A$ and a $B$, i.e., $x$ is a subtype of $A$ and of $B$
  \item and thus $x$ has both $A$’s methods and $B$’s methods
  \end{itemize}

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

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

- 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 $/\\n
- **Abbreviation:**
  - 0 or 1 occurrence

```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

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

- 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

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)
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 \( e_0.m(e_1, \ldots, e_n) \)
    - Look up \( m \) in \( e_0 \), or in classes/modules \( e_0 \) 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:()→...]

```ruby
module Kernel
  print : (*[to_s : () → String]) → NilClass
end
```
The Self Type

- 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)

```ruby
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 parameter

Type instantiation

Method polymorphism
Tuple Types

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

- \( f : () \rightarrow \text{Array<Fixnum or Boolean>} \)
  - Not precise enough to type above example
- \( f : () \rightarrow \text{Tuple<Fixnum, Boolean>} \)
  - \( \text{Tuple<t1, ..., tn>} = \text{array where elt } i \text{ has type } t_i \)
- Implicit subtyping between \text{Tuple} and \text{Array}
  - \( \text{Tuple<t1, ..., tn>} < \text{Array<t1 or ... or tn>} \)
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

```ruby
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-effect
- Tricks like this are heavily used by the `rubygems` package manager
Send and Eval

• Calling `send` performs reflective method invocation

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

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

```ruby
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

```ruby
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 1: 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 1 (cont’d)

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

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
    
    ```ruby
    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!
• 1 due to syntactic confusion

```ruby
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)

• 1 due to odd loop exit

```ruby
$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

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

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

```ruby
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

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

- 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

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

- 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

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