Safe Programming in Dynamic Languages

Jeff Foster
University of Maryland, College Park

Joint work with David An, Avik Chaudhuri, Mike Furr, Mike Hicks, Brianna Ren, T. Stephen Strickland, and John Toman

Dynamic Languages

- Dynamic languages are very popular
  - C.f. Bloomberg learning to code in JavaScript!
    - Codeacademy.com
- Dynamic languages are great for rapid development
  - Time from opening editor to successful program run is small
- Dynamic languages are convenient for big data
  - Try not to "get in the programmer's way"
  - Rich libraries, flexible syntax, domain-specific support (e.g., regexps, syscalls)
    - Can often encode "little languages" inside scripting languages

Drawbacks

- Flexible syntax can make typos suddenly meaningful

```
def foo(h1, h2) ... end  # h1, h2 hash tables
foo({:a ⇒ 10}, {:b ⇒ "foo"})  # params clear

foo :a ⇒ 10, :b ⇒ "foo" # saved some typing, but oops!
```

- Dynamic typing means type errors can remain latent until run time
  - Also, no static types to serve as (rigorously checked) documentation
  - May make code evolution and maintenance harder
- Performance a challenge
  - Dynamic typing, eval, send, method_missing, etc
  - Inhibit traditional compiler optimizations (but see JavaScript!)

Types for Ruby

- Over last several years, have been working on bringing some benefits of static typing to Ruby
  - Ruby = Smalltalk + Perl
- Goal: Make types optional and useful
  - Develop a program without types (rapidly)
  - Include them (later) to provide static checking where desired
  - Find problems as *early* as possible (but not too early!)
- Plan:
  - Discuss lessons learned from this work
  - Talk about ideas for scripting and big data

Take One: 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
  - 185 classes, 17 modules, and 997 methods (manually) typed

Intersection Types

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

- Method has all the given types
  - Ex: "foo".slice(3); "foo".slice(3..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

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 J s.t. A < J and B < J
- 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

Object Types

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

- print accepts 0 or more objects with a to_s method
  - may have other methods too
- With object types we can avoid the closed-world assumption, i.e., we don't have to write
  - print : *(C1 or C2 or ...) → %nil
    - where Ci has to_s method
- But nominal types are more terse and oftentimes more evocative, so supporting both works best

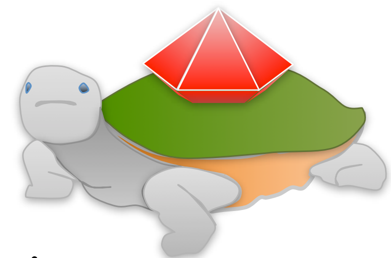
Generics: Array and Tuple Types

```
x = [ 1, 2, 3 ]
def g() [ 1, true ] end
a, b = g()                          # a = 1, b = true
```

- x : Array⟨Fixnum⟩
- g : () → Tuple⟨Fixnum, Boolean⟩
  - not () → Array⟨Fixnum or Boolean⟩
  - Tuple⟨t1, ..., tn⟩ = array where element i has type ti
- Implicit subtyping between Tuple and Array
  - Tuple⟨t1, ..., tn⟩ ≤ Array⟨t1 or ... or tn⟩

Experience (through 2010)

- We built a static inference tool for this type system
  - Diamondback Ruby (DRuby)



- Development was painstaking
  - context-sensitive parsing, surprising semantics
- Hard to support for dynamic features
  - eval, method_missing, etc.
  - Built profile-directed inference system to compensate
- Significant work to keep up to date
  - Doesn’t work with Ruby 1.9 (latest version)
- Conclusion: need lighter-weight support

Code produced at runtime

```
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
```

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

Take Two: Rubydust and Rtc

- **Ruby D**ynamic **U**nraveling of **S**tatic **T**ypes
  - Type inference
- The **R**uby **T**ype **C**hecker
  - Type checking
- Pure Ruby libraries
  - Dynamic analysis—does not examine source code
  - Infers or checks types at run time
  - Later than pure static analysis, but...
  - Earlier than Ruby's type checks

Types are Run-time Objects

- Type information is stored in class objects

```
class Array
  rtc_annotated :t
  typesig "[] : (Range) → Array<t>"
  typesig "[] : (Fixnum, Fixnum) → Array<t>"
  typesig "[] : (Fixnum) → t"
  typesig "map<u> : () {(t)→ u}→ Array<u>"
end
```

  - If generic type is instantiated, the instantiation of the type variable is stored in the constructed object

Type Wrapping

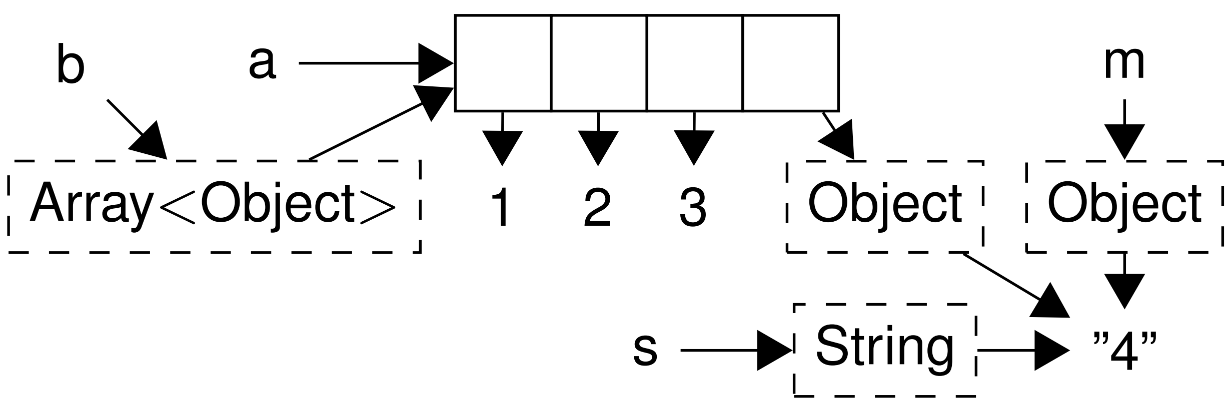
- To track type information at run-time, we wrap objects in *proxies*

```
x = 1.rtc_annotate("Fixnum")
# equivalent to...
x = Proxy.new(1,"Fixnum")
```

  - Proxied object delegates all calls to the underlying object
  - Rtc: checks types on entry and exit of method
  - Rubydust: generates type constraints on entry and exit of method
- Why is this useful:
  - Rtc: can associate a *larger* type with object than run-time type
  - Rubydust: can associate *type variable* with object

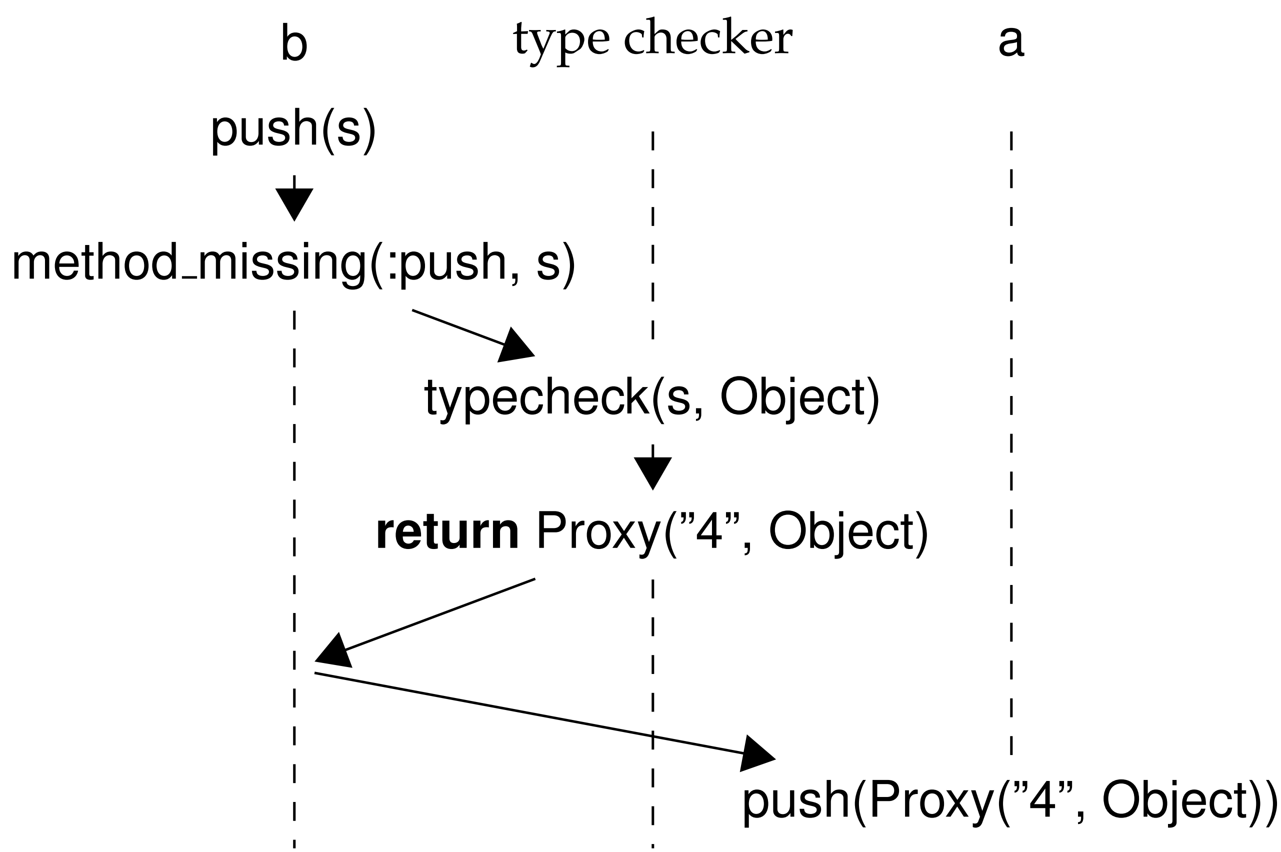
Type Wrapping Example

```
a = [1,2,3]
b = a.rtc_annotate("Array<Object>")
# Notice that b's type captures programmer intention
s = "4".rtc_annotate("String")
b.push(s)
m = b[3]
```



Proxy Calling Sequence

- b.push(s) from previous slide



Evaluation

- Ran DRuby, Rubydust, and Rtc on a range of programs
- Found lots of interesting mistakes

- Rubydust and Rtc performance acceptable on small examples, but slow
  - Worst case: Sudoku-1.4 test suite goes from 0.04s to 7.58s (rtc)
  - Lots of wrapping/unwrapping happening
  - $\Rightarrow$ Probably need to add direct interpreter support

Dynamic Languages for Big Data

- Several interesting challenges...

Correctness

- A lot of science is done by software
  - Scripting languages are increasingly popular for this
  - How do we know that software is actually computing the right results?
    - If not, conclusions may be invalid!
    - ∃ papers that have been retracted because of software bugs

- Types are a first step in helping check correctness
  - Types are very good for "computer science" software
    - Folklore: If an OCaml program type checks, it is correct
      - (N.B.: I have disproved this myself many times...)
  - What is the equivalent folklore for scientific software?

Debugging

- Suppose one of our scripts isn't working
  - How do we figure out what's wrong and fix it?
  - Are the problems in the software? In our algorithmic idea? In our scientific hypothesis?
  - Can we do better than print statements
    - Doing better is *only* important for complex bugs
    - (Simple bugs can be found with almost any approach)

- Debugging very painful for long, complex computations
  - What if the bug only manifests 1 hour in? 24 hours in?
  - Record and replay a solution?

Notation (Domain-specific Langs)

- One really nice feature of Ruby is that it makes it easy to create nice-looking DSLs in Ruby syntax

```
every :sunday, :at => "12:30am" do
  rake "talks:send_this_week"
end
```

```
  resources :lists do
    member do
      get :subscribe
      get :feed
      get :show_subscribers
    end
  end
```

- What DSLs do we want for working with big data? With bio data?
  - Is R the answer?

Performance

- Most scripting languages have poor performance
  - Ruby is known to be slow even without proxies/wrapping
    - Improved significantly in 1.9, but still not great
  - Python is a memory hog
    - Based on our experience using Python in debugging large systems
  - Exception: Lua is quite zippy
    - But it doesn't have some of the nice features of Ruby and Python
- We need to do better to work with big data sets
  - In JavaScript, trace-based just-in-time compilation is hot
    - A key transformation: specialization based on types
      - cf. David Padua's talk yesterday—ROpt
    - Do the same ideas work in Python, Ruby, and R?

Updates

- What happens if we start a long computation running, and then halfway through we want to change it?
  - E.g., found minor bug that could be worked around
  - E.g., found performance problem we want to fix

- Dynamic software updating
  - Change code and data representations on the fly
    - Support for *state transformation* to change old state to new form
  - Research to date has focused on operating systems and on long-lived servers
  - Investigate for big data programs?

Program Synthesis

- Old idea: given a *specification*, automatically synthesize a program that satisfies the specification
  - New energy: SMT solver and other algorithmic performance improvements have made this possible, at least in the small
- Recent success stories
  - Synthesizing FFTs that out-perform hand-coded implementations
  - Synthesizing synchronization placement in high-performance code
  - Synthesizing Excel macros
- Apply to big data / bio domains?
  - Can we use synthesis to create an even higher-level way of describing big data algorithms? Can we find new algorithms this way?

Possible Topics

- Correctness
- Debugging
- Notation (DSLs)
- Performance
- Updates
- Program Synthesis