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

```ruby
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

• 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

- 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

```
class A def f() end end
class B def f() end end
x = ( if ... then A.new else B.new)
x.f
```
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 : () \rightarrow Tuple⟨Fixnum, Boolean⟩
  - not () \rightarrow Array⟨Fixnum or Boolean⟩
  - Tuple⟨t₁, ..., tn⟩ = array where element i has type tᵢ
• Implicit subtyping between Tuple and Array
  - Tuple⟨t₁, ..., tn⟩ \leq 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
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

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

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

Huh?
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
  ...

Read the current file
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
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

```ruby
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  
...
Another Fun Example

```ruby
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

```ruby
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
...
```
Take Two: Rubydust and Rtc

- **Ruby Dynamic Unraveling of Static Types**
  - Type inference
- The **Ruby Type Checker**
  - 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

```ruby
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]

---

Figure 4: Wrapping in rtc

Preliminary Work

The key idea behind RubyDust and rtc is to maintain type information (types as in Figure 2) at run time and check those types whenever we cross an abstraction barrier, in particular, at entrance and exit to methods.

We illustrate this idea by discussing type checking in rtc; type inference in RubyDust is similar, but it generates typing constraints wherever rtc would perform a check and then solves those constraints at the end of execution.

Consider the code in Figure 4a, which generates the heap structure shown at the bottom of the figure. To track type information, rtc wraps type-annotated objects with proxy objects that hold types. When methods are invoked on proxies, rtc performs type checking before and after delegating to the underlying method.

Wrapping in rtc lets us type an object with a bigger type than its run-time class (examples below); wrapping in RubyDust lets us associate a type variable with an object.

We begin on line 94 by creating an array, which we then annotate with type Array<Object> on line 95; the annotated type is represented by an instance of rtc's Proxy class. Thus, rtc distinguishes raw objects, such as a, from annotated objects, such as b. In rtc, annotated objects form the "roots" of type checking, so that programmers can continue to leave portions of the program unannotated as needed [65]. Notice also that the type annotation on line 95 is larger than what is apparent from the current contents of a; thus, type annotations provide an important indication of (non-obvious) programmer intent.

Next, line 97 assigns a new Proxy to s. Then consider the call on line 98; the events triggered by this call are shown in Figure 4b. When push is invoked on the proxy object b, it is intercepted by the proxy's method missing method, which receives calls to any undefined methods. In turn, that method checks the argument type (more precisely, it checks if String is a subtype of Object, the array contents type); rewraps s to associate it with the formal argument type Object; and then passes that wrapped object into Array's push method. Thus, when we retrieve s from the array on line 100, it is typed as Object, which captures the programmer's intention from line 95.

Proposed Work

We propose to extend the core approach of rtc and RubyDust to work with the specifications proposed in Section 3. Notice that this approach also meshes well with our design of writing specifications, like those in Figure 3, in executable Ruby code. There are several challenges in achieving our goal:

Handling additional features of Ruby and Rails

While rtc and RubyDust are compelling prototypes, they have a number of basic limitations we must address. At the core language level, they lack full support
Proxy Calling Sequence

- `b.push(s)` from previous slide

```
b         type checker         a
push(s)\nmethod_missing(:push, s)\ntypecheck(s, Object)\nreturn Proxy("4", Object)\npush(Proxy("4", Object))
```

Preliminary Work

The key idea behind RubyDust and rtc is to maintain type information (types as in Figure 2) at run time and check those types whenever we cross an abstraction barrier, in particular, at entrance and exit to methods. We illustrate this idea by discussing type checking in rtc; type inference in RubyDust is similar, but it generates typing constraints wherever rtc would perform a check and then solves those constraints at the end of execution.

Consider the code in Figure 4a, which generates the heap structure shown at the bottom of the figure. To track type information, rtc wraps type-annotated objects with proxy objects that hold types. When methods are invoked on proxies, rtc performs type checking before and after delegating to the underlying method.

Wrapping in rtc lets us type an object with a bigger type than its run-time class (examples below); wrapping in RubyDust lets us associate a type variable with an object.

We begin on line 94 by creating an array, which we then annotate with type `Array<Object>` on line 95; the annotated type is represented by an instance of rtc's `Proxy` class. Thus, rtc distinguishes raw objects, such as `a`, from annotated objects, such as `b`. In rtc, annotated objects form the "roots" of type checking, so that programmers can continue to leave portions of the program unannotated as needed [65]. Notice also that the type annotation on line 95 is larger than what is apparent from the current contents of `a`; thus, type annotations provide an important indication of (non-obvious) programmer intent.

Next, line 97 assigns a new `Proxy` to `s`. Then consider the call on line 98; the events triggered by this call are shown in Figure 4b. When `push` is invoked on the proxy object `b`, it is intercepted by the proxy's `method_missing` method, which receives calls to any undefined methods. In turn, that method checks the argument type (more precisely, it checks if `String` is a subtype of `Object`, the array contents type); rewraps `s` to associate it with the formal argument type `Object`; and then passes that wrapped object into `Array`'s `push` method. Thus, when we retrieve `s` from the array on line 100, it is typed as `Object`, which captures the programmer's intention from line 95.

Proposed Work

We propose to extend the core approach of rtc and RubyDust to work with the specifications proposed in Section 3. Notice that this approach also meshes well with our design of writing specifications, like those in Figure 3, in executable Ruby code. There are several challenges in achieving our goal:

- Handling additional features of Ruby and Rails

While rtc and RubyDust are compelling prototypes, they have a number of basic limitations we must address. At the core language level, they lack full support
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
  ▪ ⇒ 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

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

```ruby
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