Static Typing for Ruby on Rails

Jong-hoon (David) An
Computer Science Department
University of Maryland
College Park, USA
davidan@cs.umd.edu

Avik Chaudhuri
Computer Science Department
University of Maryland
College Park, USA
avik@cs.umd.edu

Jeffrey S. Foster
Computer Science Department
University of Maryland
College Park, USA
jfoster@cs.umd.edu

***Abstract*—Ruby on Rails (or just "Rails") is a popular web application framework built on top of Ruby, an object-oriented scripting language. While Ruby's powerful features such as dynamic typing help make Rails development extremely lightweight, this comes at a cost. Dynamic typing in particular means that type errors in Rails applications remain latent until run time, making debugging and maintenance harder. In this paper, we describe DRails, a novel tool that brings static typing to Rails applications to detect a range of run time errors. DRails works by translating Rails programs into pure Ruby code in which Rails's numerous implicit conventions are made explicit. We then discover type errors by applying DRuby, a previously developed static type inference system, to the translated program. We ran DRails on a suite of applications and found that it was able to detect several previously unknown errors.**

***Keywords*-Ruby; Ruby on Rails; scripting languages; type systems; web frameworks**

I. Introduction

Web application frameworks have become indispensable for rapid web development. One very popular framework is Ruby on Rails (or just "Rails"), which is built on top of Ruby, an object-oriented scripting language. While Ruby allows development in Rails to be extremely lightweight, Ruby's dynamic typing means that type errors in Ruby programs, and hence Rails programs, can remain latent until run time. Recently, we have been developing Diamondback Ruby (DRuby), a new static type inference system for ordinary Ruby code [1], [2]. We would like to bring the same type inference to Rails to catch common programming bugs in Rails programs.

Unfortunately, by itself, DRuby would be essentially useless on Rails code. The Rails framework uses a significant amount of highly dynamic, low-level class and method manipulation as part of its "convention over configuration" [3] design. This kind of code is essentially unanalyzable with DRuby (and with any typical static analysis). We also cannot simply omit the framework during analysis of an application, since we would then miss most of the application's behavior.

In this paper, we address this issue with DRails, a novel tool that brings DRuby's type inference to Rails. The key insight behind DRails is that we can make *implicit* Rails conventions *explicit* through a Rails-to-Ruby transformation, and then analyze the resulting programs with DRuby. Type errors in the transformed programs indicate type errors in the original Rails applications. As far as we are aware, DRails is the first tool to bring static typing to Rails. Furthermore, we expect that DRails's transformation can serve as a front-end for other static analyses on Rails programs, and the idea of analyzing programs by transformation can be applied to other code development frameworks as well.

We evaluated DRails by running it on a suite of 11 Rails programs gathered from a variety of sources. DRails found 12 previously unknown errors that can cause crashes or unintended behavior at run time. DRails also identified 2 examples of questionable coding practice. The fact that DRails could find these errors is particularly surprising since Rails applications are often thoroughly tested during development using Rails's built-in testing infrastructure. Furthermore, DRails reported 57 false positives; about half of them were due to known incompleteness issues in DRuby, and we expect most of the others to be eliminated with minor extensions to DRails.

II. Overview

Rails is built on top of Ruby, an object-oriented scripting language [4]. Rails uses a *model-view-controller* (MVC) architecture [5], in which any web request by the client results in a call to some method in a *controller*, which in turn uses a *model* to perform database accesses and eventually returns a *view*, i.e., the text of a web page, as the response. To illustrate how Rails works and the challenges of reasoning about Rails applications, we will develop a small program called *catalog* that maintains an online product catalog. The database for *catalog* tracks a set of companies, each of which has a set of products. In turn, each product has a name plus a longer textual description. Internally, *catalog* has two models (Company and Product), two controllers (CompaniesController and ProductsController), and one view (views/companies/info.html.erb). We next discuss the code for these various components, potential bugs that might appear in this simple application, and how DRails would help detect these errors statically. Due to space constraints, our discussion is incomplete; more details can be found in our companion technical report [6].

db/schema.rb

```
1  create_table "companies" do |t| t.string "name" end
2  create_table "products" do |t|
3    t.integer "company_id"
4    t.string "name"
5    t.string "description"
6  end
```

models/company.rb

```
7  class Company < ActiveRecord::Base
8    has_many :products
9    validates_uniqueness_of :name
10 end
```

controllers/companies_controller.rb

```
11 class CompaniesController < ActionController::Base
12   def info()
13     @company = Company.find_by_name (params[:name])
14   end
15 end
```

views/companies/info.html.erb

```
16 <h2><%= @company.name %></h2>
17 <% @company.products.each do |product| %>
18   <p><%= product.name + ":" + product.description%></p>
19 <% end %>
```

controllers/products_controller.rb

```
20 class ProductsController < ActionController::Base
21   before_filter :authorize, :only ⇒ :change
22   def info
23     company = Product.find(params[:id]).company
24     redirect_to :controller ⇒ "companies", :action ⇒ "info",
25       :name ⇒ company.name
26   end
27   def change
28     @product.description = params[:description]
29     @product.save
30     info
31   end
32   private
33   def authorize
34     @product = Product.find(params[:id])
35     return @product.company.name == session[:user] ? nil : info
36   end
37 end
```

Figure 1. Rails application *catalog* source code

A. Models

The first listing in Fig. 1 shows db/schema.rb, which is a Ruby file that is auto-generated from the database table. (The code for a Rails application is split across several subdirectories, including db/ for the database, and models/, views/, and controllers/ for the correspondingly named components.) This file records the names of the tables and the fields of each row: the companies table has a name field, and the products table has name and description.

In Rails, each row in a table is mirrored as an instance of a *model* class (or, just "model"), which must be defined by a file in the models/ directory. The second listing in Fig. 1 shows the Company class, corresponding to the companies table. (We omit the Product class from the figure due to lack of space.) Note the singular/plural relationship between model (Company) and table (companies) names. Rails uses the information from schema.rb to automatically add field setter and getter methods to the models, among other things. For example, it creates methods name() and name=() to Company to get and set the corresponding field name.

Because these and other methods are created implicitly by Rails, and since Ruby has no static type checking, it is easy to make a mistake in calling such a method and not realize it during development. In DRails, we transform the initial program, explicitly generating Ruby code corresponding to auto-generated methods. For example, the Company model is modified as follows:

```
class Company < ActiveRecord::Base
  attr_accessor :id, :name
  ...
```

The calls to attr_accessor create methods to read and write fields @id and @name. We pass the transformed code to DRuby, which can then check that uses of these accessors are type correct.

Models not only have methods added to them based on the database schema, but they also inherit from the Rails class ActiveRecord::Base (as shown on line 7; < indicates inheritance). This class defines a variety of useful methods, including several that tell Rails about relationships between tables. In our example, each company can have many products, indicated by the call on line 8, which adds methods products() and products=() (note the pluralization) to Company. For these methods to function, Rails requires that the company_id field declared on line 3 exist.

To type programs that use this feature or similar features, DRails needs to add the implied method definitions and the implied calls to the program. For example, the **has_many** call on line 8 is transformed into the following set of type annotations:

```
class Company < ActiveRecord::Base
  ##% products : () → HasManyCollection<Product>
  ...
  ##% products= : \
  ##%   (Array<Product>) → HasManyCollection<Product>
  ...
end
```

Here the getter method products returns a collection of Product objects. The setter method products= takes an array of Products objects and returns the updated collection.

Next, if a model instance is updated or created, the save() method (inherited from ActiveRecord::Base) is called to commit it to the database. This method will reject objects whose *validation* methods fail. For example, line 9 calls **validates_uniqueness_of** :name to require the name

field of a company to be unique across all companies. Programmers can also define custom validation methods that include arbitrary Ruby code, but we have omitted this due to space limitations. DRails ensures that such behaviors are correctly captured in the analysis by inserting explicit calls to validation methods in appropriate places.

There are also a few other implicit model conventions that DRails makes explicit. One important case is find_by_x(y), which, if called, returns the first occurrence of a record whose x field has value y, as shown in line 13 of Fig. 1. There is one such method, plus one find_by_all_x method, for each possible field. DRails adds type annotations for these methods to the model, e.g., since Company has a field name, DRails adds annotations for find_by_name and find_all_by_name to class Company.

B. Controllers and Views

In Rails, the *actions* available in a web application are defined as methods of *controller* classes. The third listing in Fig. 1 shows CompaniesController, which, as do other controllers, inherits from ActionController::Base. This controller defines an action info that allows clients to list the products belonging to a particular company. This action is invoked whenever the client requests a URL beginning with "⟨server⟩/companies/info", and it expects a parameter name to be passed as part of the POST or GET request. When info is called, it finds the Company row whose name matches params[:name], the requested name, and stores it in field @company (line 13). The last step of an action is often a call to **render**, which displays a view. In this case, info includes no such call, so Rails automatically calls **render** :info to display the view with the same name as the controller.

The corresponding view is shown as the fourth listing in Fig. 1. As is typical, this view is written as an .html.erb file, which contains HTML with embedded Ruby code. Here, text between <% and %> is interpreted verbatim as Ruby code, and text between <%= and %> is interpreted as a Ruby expression that produces a string to be output in the resulting web page. For example, line 16 shows a second-level heading whose content is the value of @company.name; recall @company was set by the controller, so it is an interesting design decision that Rails allows it to be accessed here. Similarly, lines 17–19 contain Ruby code to iterate through the company's products and render each one.

The last listing in Fig. 1 defines a more complex controller, ProductsController, with several actions. The first one, info (lines 22–26), computes the company of the product given by the parameter id and then uses **redirect_to** to pass control to the info action of CompaniesController (lines 11–15), specifying the company's name. As we discussed above, this in turn calls **render** :info (lines 16–19). It is possible to call **redirect_to** several times before eventually calling **render**, allowing control to flow through several controllers before eventually displaying a view.

The change action (lines 27–31) allows a product description to be updated. However, we only want to allow authorized users to make such changes. Thus, on line 21 we call **before_filter** to specify that the authorize action should always be run before change. Note that authorize is declared **private** (line 32), so it cannot be called directly as an action.

When authorize is called, it looks up the product to be modified (line 34) and checks whether the user logged into the current session (stored in session[:user]; this is established elsewhere (not shown)) matches the name of the company of that product (line 35). If so, then authorize evaluates to **nil**, and control passes to change, which updates the product description (line 28), commits the change to the database (line 29), and then calls info to show the product listing screen (line 30). Otherwise, authorize calls info, and since that ends in a **redirect_to**, the action change will never be rendered.

Like models, controllers and views can have errors that are hard to detect. First, view file names could have the wrong extension, in which case Rails may be unable to find them, causing crashes or unintended behavior. Second, a (perhaps implicit) call to **render** could go to a non-existent view. Third, as control flows get complex, with actions inserted before other actions with filters, and actions in one controller calling actions in another, it is easy to make a typo in the method name for a filter. For example, writing :authorized instead of :authorize on line 21 will crash the program. Or, making a mistake in a **redirect_to** call (say, by writing "company" instead of "companies" on line 24, or @company = ... rather than company = ... on line 13) will also result in an unexpected behavior.

Analogously to the transformation of models, DRails performs code transformation for views and controllers. To fully reason about views, we first need to able to analyze the Ruby code embedded in HTML. Our solution is to use Markaby for Rails [7] to parse the views and produce regular Ruby classes that generate the same dynamic web pages. (Note that while Markaby worked as-is on small examples, we needed to make major improvements to apply it to our suite of programs in Section III.) We call this process *Rubyifying* the view. For example, here is the result of Rubifying views/companies/info.html.erb of Fig. 1, slightly simplified for discussion purposes:

```
module CompaniesView
  include ActionView::Base
  def info
    Rubify.h2 do Rubify.text(@company.name) end
    @company.products.each do |product|
      Rubify.p do
        Rubify.text(product.name + ":" + product.description)
end end end end
```

Here the method name info is based on the view name info. The calls to Rubify's methods output strings containing the appropriate HTML; notice that the calls are intermixed with

regular Ruby code. For example, Rubify.h2 **do**... **end** creates the second-level heading on line 16 of Fig. 1. We created this method as part of **module** CompaniesView, where the module name was derived from the file's location under views/. Rails does approximately the same thing, implicitly creating a CompaniesView class from the view.

Summing up, even an application as simple as *catalog* contains many opportunities for error. DRails can find many Rails errors by transforming the original program and running DRuby on the result.

III. Implementation and Experiments

DRails comprises approximately 1,700 lines of OCaml and 2,000 lines of Ruby. DRails begins by combining all the separate files of the Rails application into one large program. DRails parses the program into the Ruby Intermediate Language (RIL), a subset of Ruby that is designed to be easy to analyze and transform [8]. Then DRails instruments this program to capture arguments passed to Rails API calls. The program is loaded with Ruby, and the resulting instrumentation output is fed back into DRails and used to transform the combined program, making uses of Rails's conventions explicit. This transformed program is passed to DRuby along with base.rb, a file that gives type signatures to remaining Rails API methods, and stub files containing type signatures for any external libraries. DRuby performs type inference and emits warnings for any errors it finds.

We evaluated DRails by running it on 11 Rails applications that we obtained from various sources including RubyForge and OpenSourceRails. Fig. 2 summarizes our results. The first group of columns gives the size of each application, in terms of source code lines (counted with wc); the size in kilobytes of the RIL control-flow graph after parsing the model, controllers, and similar files and Rubifying the views; and the size in kilobytes of the RIL control-flow graph after full transformation. This increase shows that there is a significant amount of code that Rails produces by convention.

Due to current limitations of DRails, we needed to make some small changes to the applications. We manually closed unbalanced HTML tags so that Rubified code is well-formed (R), flattened nested directory structures to avoid confusion in matching class names to files (H), transformed non-literal arguments to **render** and **redirect_to** into **case** statements to capture all routing behaviors (I), and manually added required files to config/environment.rb (B).

The results of running DRails on these programs are tabulated in the last two groups of columns in Fig. 2. We ran DRails on an AMD Athlon 4600 processor with 4GB of memory. We break down the running times of DRails into DRails-only time, DRuby time, and the total time. The reported running times are the average of three runs. The DRails-only step is fairly fast across all the applications, and most of the running time is due to DRuby.

We manually categorized DRuby's error reports into four categories: *errors* (E), reports that corresponds to bugs that may crash the program at run time or cause unintentional behavior; *warnings* (W), reports for code that behaves correctly at run time, but uses suspicious programming practice; *deprecated* (D), reports of uses of Rails features no longer available in Rails 2.x; and *false positives* (F) that do not correspond to actual bugs.

We found 12 errors in the applications. Eight of the errors, six in *lohimedia* and two in *onyx*, are due to programmer misunderstandings of Ruby's syntax. For example, *lohimedia* contains the code:

```
flash[:notice] = "You do not have..."
+ "..."
```

Here the programmer intends for the string on the second line to be concatenated with the first line. In Ruby, however, line breaks affect parsing, so the string on the first line is assigned to flash[:notice]. Then the second line results in a call to the unary method + with a string argument, which is a type error. Because Ruby is dynamically typed, errors like this can remain latent until run-time, whereas DRuby (and DRails) can find such bugs statically.

The other two errors in *onyx* are due to the following embedded Ruby code:

```
<% @any_more = Post.find(:first, :offset ⇒ (@offset.to_i +
      @posts_per_page.to_i) + 1, :limit ⇒ 1 ) %>
```

Here DRuby reports that Post, which the programmer seems to be treating as a model, is undefined, as indeed it is.

One error in *diamondlist* is due to invoking the nonexistent method ≪ on a Hash. (A method with that name does exist in Array, perhaps explaining the error.) The other error in *diamondlist* occurs in call to **render** in which the specified view, top_bar, does not exist.

Finally, *boxroom* has an interesting error in one of its models due to a call to an undefined method password_confirmation. This method name is commonly used by convention in Rails applications, but it is only available if the user declares both password and password_confirmation fields, usually by calling attr_accessor. However, in this case the programmer instead calls attr_accessible on these fields, which has completely different semantics.

We found 2 warnings in total. The first warning is due to a call to a method that is not in Rails documentation but actually part of a Rails definition. We are not sure whether this method should have been documented or is meant to be private. The second warning occurs due to a method call whose block argument has the wrong type signature [1]. We found 72 uses of deprecated constructs that operate correctly on older versions of Rails but cause run-time errors on Rails 2.x. benchmarks. DRails reported 57 false positives due to: limitations in DRuby's annotation language; features not handled by DRails; inconsistent variable scoping due to

		CFG sizes (kb)		Patches (#)				Running times (s)			Errors (#)			
	LoC	Before	After	R	H	I	B	DRails	DRuby	Total	E	W	D	F
depot	997	139	358	·	·	·	1	2.30	9.74	12.04	·	1	1	1
moo	838	143	402	4	·	·	3	2.45	18.76	21.21	·	·	·	3
pubmgr	943	196	548	·	·	·	1	3.00	26.41	29.41	·	·	·	·
rtplan	1,480	273	697	·	·	·	2	3.47	26.65	30.12	·	·	6	1
amethyst	1,183	264	729	·	·	·	4	3.53	39.03	42.56	·	·	·	1
diamondlist	1,415	265	786	4	2	·	1	4.10	23.81	27.91	2	·	·	2
chuckslist	1,447	329	883	1	·	9	4	4.08	52.23	56.31	·	1	2	14
boxroom	2,330	376	959	6	·	1	2	4.16	87.23	91.39	1	·	27	6
onyx	2,228	484	1,190	6	1	·	1	5.62	79.75	85.37	3	·	·	1
mystic	2,822	639	1,525	13	·	5	1	6.38	146.40	152.78	·	·	·	11
lohimedia	11,106	1,290	3,331	9	·	2	3	14.01	662.95	676.96	6	·	36	17

Figure 2. Experimental results

newly introduced Ruby code during Rubification; and run-time type tests that DRuby cannot analyze.

Threats to Validity We should emphasize that DRails by no means checks for all possible errors in Rails programs, e.g., clearly Rails programs can have errors that are unrelated to types. Beyond that, there are several potential threats to the validity of our experimental results. First, our type signatures for the Rails API could be overly general, allowing calls that might fail at runtime. Second, DRails's modeling of the Rails API is incomplete and could be slightly inaccurate. Third, our categorization of some of DRails's errors might be incorrect, e.g., we may have classified code as erroneous that actually behaves correctly at runtime. Finally, there could be bugs in DRuby that cause it to unsoundly miss type errors.

IV. Related Work

Most existing work on static analysis of web applications focuses on verification of security properties. Lam et al. [9] combine static analysis with model checking to verify that information-flow patterns are satisfied in Java-like programs. Huang et al. [10] use a lattice-based static analysis algorithm derived from type systems and typestate to ensure similar information-flow properties. The tool TAJ [11] performs taint analysis of web applications written in Java, and uses novel program slicing techniques to handle reflective calls and flows through containers. The tool Pixy [12] performs alias analysis for PHP and finds security vulnerabilities in web applications written in PHP. Xie and Aiken [13] address the same problem, and present a static analysis algorithm based on symbolic evaluation to handle dynamic features of PHP. The key differences between all of these systems and DRails is our focus on static typing and Ruby on Rails, a combination we believe we are the first to study.

V. Conclusion

In this paper, we presented DRails, a novel static analysis tool for Rails applications. DRails works by translating Rails applications into pure Ruby code in which the automation provided by Rails's sophisticated internal machinery is made explicit. We then apply DRuby, a static type inference system for Ruby, to the result. We showed that static typing catches a variety of bugs in Rails applications, and we believe we are the first to bring static typing to Rails.

References

[1] M. Furr, J. An, J. S. Foster, and M. Hicks, "Static Type Inference for Ruby," in *OOPS Track, SAC*, 2009.

[2] M. Furr, J. An, and J. S. Foster, "Profile-guided static typing for dynamic scripting languages," in *OOPSLA*, 2009, to appear.

[3] "Ruby on Rails," 2009, http://rubyonrails.org.

[4] D. Flanagan and Y. Matsumoto, *The Ruby Programming Language*. O'Reilly Media, Inc, 2008.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*. Addison-Wesley, 1995.

[6] J. An, A. Chaudhuri, and J. S. Foster, "Static typing for ruby on rails," University of Maryland, College Park, Tech. Rep., 2009, http://www.cs.umd.edu/~davidan/papers/drails.pdf.

[7] "Markaby for Rails," 2006, http://redhanded.hobix.com/inspect/MarkabyforRails.html.

[8] M. Furr, J. An, J. S. Foster, and M. Hicks, "The ruby intermediate language," in *DLS*, 2009, to appear.

[9] M. S. Lam, M. Martin, B. Livshits, and J. Whaley, "Securing web applications with static and dynamic information flow tracking," in *PEPM*, 2008, pp. 3–12.

[10] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo, "Securing web application code by static analysis and runtime protection," in *WWW*, 2004, pp. 40–52.

[11] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman, "TAJ: Effective taint analysis for Java," in *PLDI*, 2009, to appear.

[12] N. Jovanovic, C. Kruegel, and E. Kirda, "Precise alias analysis for static detection of web application vulnerabilities," in *PLAS*, 2006, pp. 27–36.

[13] Y. Xie and A. Aiken, "Static detection of security vulnerabilities in scripting languages," in *USENIX Security*, 2006, pp. 179–192.