Static Typing for Ruby on Rails

Jong-hoon (David) An, Avik Chaudhuri, and Jeffrey S. Foster
Computer Science Department, University of Maryland, College Park
Email: {davidan,avik,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 help make Rails development extremely lightweight, this comes at a cost: Ruby is dynamically typed, and so type errors in a Rails application can 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.

1. Introduction

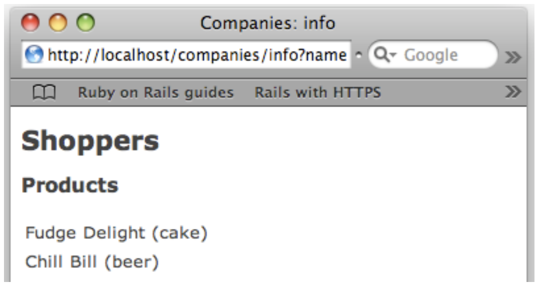
Web application frameworks have become indispensable for rapid web development. One very popular framework is Ruby on Rails (or just "Rails"), which is built on top of Ruby, an object-oriented scripting language. While Ruby allows Rails development to be extremely lightweight, it also introduces a significant challenge. Ruby is dynamically typed, and that means that type errors in Ruby programs, and hence Rails programs, can remain latent until run time. Our main observation in this paper is that many common programming bugs in Rails programs are essentially due to such type errors. To give some Rails-specific examples, the programmer could make a typo when referring to a database table, could call a non-existing field accessor method, or could make type errors in Ruby code embedded inside HTML. Anecdotally, the lack of static types can also impede maintainability, and means that programmers miss out on the automatically enforced documentation that types can provide.

Recently, we have been developing Diamondback Ruby (DRuby), a new static type inference system for ordinary Ruby code [6, 5]. 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, for two reasons.

The first problem is that Rails favors "convention over configuration" [19], so that analyzing only the application code would be insufficient. For example, suppose that an application uses a database table called `students`. Rails will automatically abstract rows of this table as instances of a Ruby class `Student`, and Rails will create accessor methods in `Student` to get and set fields according to the database schema. While such a design leads to very concise code, it makes Rails programs unanalyzable with DRuby, or indeed with most other static analyses—there are too many implicitly created methods, which DRuby would think are missing; too many conventions relating names in different parts of the application, which DRuby would fail to check; and too many implicit method calls, which DRuby would not see, and hence would not type check. (More examples of this problem appear in Section 2.)

The second problem is that even if we included the framework code (which implements the conventions) in our analysis, the resulting code would still be unanalyzable by DRuby. Indeed, this code uses highly dynamic, low-level class and method manipulations that are typically hard to analyze statically.

In this paper, we address these problems 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


Companies: info

http://localhost/companies/info?name

Ruby on Rails guides Rails with HTTPS

Shoppers

Products

Fudge Delight (cake)

Chill Bill (beer)

Figure 1. Screenshot from *catalog*

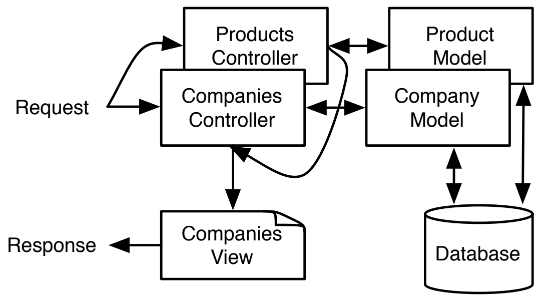


Figure 2. Rails MVC architecture

analyzing programs by transformation can be applied to other code development frameworks as well.

DRails's transformation itself is fairly complicated, because Rails has many moving parts. The major steps include parsing a Ruby file containing the database schema; transforming HTML files with embedded Ruby into pure Ruby code that renders the same web page; using a dynamic "load-time" analysis to discover how the Rails application calls the Rails API; and finally inserting the implied method definitions and calls into the source code. Some of our implementation details are interesting in their own right, as they allow us to optimize our transformation code and provide more assurance of the faithfulness of the transformed code (Section 3).

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 such applications are often thoroughly tested during development using Rails's in-built 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 (Section 4).

We believe these results suggest that DRails is a promising new tool for preventing errors in Rails applications, and we think that our transformation-based approach will prove very useful for many other future static analyses for Ruby on Rails.

2. Reasoning About Ruby on Rails

Rails is built on top of Ruby, an object-oriented scripting language [4]. 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. As we will see in the following sections, the small size of the program is somewhat illusory; there is a lot of implicit code run by Rails even for this program, and such code typically blows up the size of programs by a factor of around 2.7 in our experiments.

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. The capabilities provided by *catalog* are illustrated with the screenshot in Figure 1. This page is generated when the user visits "⟨server⟩/companies/info?name=Shoppers", and it shows the products belonging to company "Shoppers." In this case, there are two products, "Fudge Delight" with description "cake," and "Chill Bill" with description "beer." The *catalog* application also allows the user to update product descriptions, and it displays the product listing screen for the owning company afterward. For example, assuming "Fudge Delight" has id 4 in the database, then if the user visits "⟨server⟩ /products/change/4?description=cake", the description for "Fudge Delight" will be updated to "cake," and the screenshot in Figure 1 will be displayed.

Rails applications use a *model-view-controller* (MVC) architecture [7], in which any web request by the client is translated into 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. Figure 2 shows how various components of *catalog* interact. A request to *catalog* eventually produces a response after possible interactions with a database. Internally, *catalog* has two models (Company and Product), two controllers (CompaniesController and ProductsController), and one view ("companies/view/info.html.erb"). A request generates a call to one of the controller actions, which possibly interacts with the database through the models, and eventually calls the view action to generate a response. We next discuss the code for these various components.

db/schema.rb

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

models/company.rb

```
10 class Company < ActiveRecord::Base
11   has_many :products
12   validates_uniqueness_of :name
13 end
```

models/product.rb

```
14 class Product < ActiveRecord::Base
15   belongs_to :company
16   validate :unique_name_in_company
17
18   def unique_name_in_company?(x)
19     x.company != company ||
20     x.name != name
21   end
22
23   def unique_name_in_company
24     Product.all.forall do |p|
25       p.unique_name_in_company?(self)
26     end
27   end
28 end
```

Figure 3. *catalog* schema and models

2.1. Models

Recall that the *catalog* application includes two database tables, one for the companies and one for their products. The first listing in Figure 3 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 fields company_id, name, and description. (A few other, minor details of this file are omitted for simplicity.)

In Rails, each row in a table is mirrored as an instance of a *model* class (henceforth, just "model"), which must be defined by a file in the models/ directory. The bottom two listings in Figure 3 show the Company class, corresponding to the companies table, and the Product class, corresponding to the products table. Note the singular/plural relationship between model and table 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() (called on line 20) and name=() to get and set the corresponding field of a Product object.

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 lines 10 and 14; < indicates inheritance). This class defines a variety of useful methods, including several that tell Rails about relationships between tables. In our example, each Product is owned by some Company, and this is indicated on line 15 by calling the (inherited) **belongs_to** method with the argument :company (a symbol). When Rails sees this call, it adds methods company() and company=() to Product. Analogously, each company can have many products, indicated by the call on line 11, 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 6 exist; this field maps each product to a company.

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 12 calls **validates_uniqueness_of** :name to create a validation method that requires the name field of a company is unique across all companies.

Programmers can also define custom validation methods that include arbitrary Ruby code. For example, line 16 registers the validation method defined in lines 23–27. This method iterates through all Products in the database (line 24) and, for each one, calls its unique_name_in_company? method with argument **self**. (Note this method's name differs from the previous one only in the trailing ?.) This method, defined on lines 18–21, returns false if the argument has the same company and name as the receiver.

Possible Errors Caught by DRails. Models already provide a rich source of errors that DRails can catch:

- Pluralization of model names is implicit in Rails, and misunderstandings of this convention can lead to hard-to-understand bugs. Even worse, having a model with a singular name foo and a model with its plural foos (or however it is inflected) can cause a lot of confusion, because Rails will map both to the table foos (as the plural of foos is foos). DRails checks for these kinds of bugs, and makes sure all the models exist as database tables.

- Various methods for accessing database columns 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. Worse, there are some idiosyncrasies in Rails's method generation that programmers might not be aware of, leading to mistakes. For example, Rails names join tables using a combination of the names of the associated tables, and the exact combination is sometimes difficult to predict. DRails helps ameliorate such problems by explicitly generating Ruby code corresponding to auto-generated methods and then using DRuby to check that method calls are type correct.
- DRails makes sure the bodies of all programmer-defined methods are type safe. For example, if on line 25 the programmer forgets to pass an argument to unique_name_in_company?, or calls unique_name_in_company instead, DRails reports that it cannot find an instance method in class Product with the required signature. As another example, if the programmer moves the || from the end of line 19 to the beginning of line 20 (a common mistake in Ruby, due to line breaks acting as statement delimiters), DRails reports that while || is expected to take 2 arguments, it only takes 1 argument here.

2.2. Controllers and Views

Moving on with our example, now that we have created our models, we can construct the actual web application. In Rails, the *actions* available in a web application are defined as methods of *controller* classes. The first listing in Figure 4 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 31). The find_by_name method called here is implicitly added to the Company model by Rails. 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.

That view, which corresponds to the screenshot in Figure 1, is shown as the second listing in Figure 4. As is typical, the view is written as an .html.erb file, which

controllers/companies_controller.rb

```
29 class CompaniesController < ActionController::Base
30   def info
31     @company = Company.find_by_name (params[:name])
32   end
33 end
```

views/companies/info.html.erb

```
34 <h2><%= @company.name %></h2>
35 <h3>Products</h3>
36 <table>
37 <% @company.products.each do |product| %>
38   <tr><td>
39     <%= product.name %> (<%= product.description %>)
40   </td></tr>
41 <% end %>
42 </table>
```

controllers/products_controller.rb

```
43 class ProductsController < ActionController::Base
44   before_filter :authorize, :only ⇒ :change
45
46   def info
47     company = Product.find(params[:id]).company
48     redirect_to :controller ⇒ "companies", :action ⇒ "info",
49       :name ⇒ company.name
50   end
51
52   def change
53     @product.description = params[:description]
54     @product.save
55     info
56   end
57
58   private
59   def authorize
60     @product = Product.find(params[:id])
61     if @product.company.name == session[:user] then nil
62     else info
63     end
64   end
65 end
```

Figure 4. *catalog* controllers and views

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 34 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 37–43 contain Ruby code to iterate through the company's products (line 37) and render each one (line 39).

The third listing in Figure 4 defines a more complex controller, ProductsController, with several actions.

The first one, info (lines 46–50), 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 30–32), specifying the company's name. As we discussed above, this in turn calls **render** :info (lines 34–42). It is possible to call **redirect_to** several times before eventually calling **render**, and it allows control to flow through several controllers before eventually displaying a view.

The change action (lines 52–56) allows a product description to be updated. However, we only want to allow authorized users to make such changes. Thus, on line 44 we call **before_filter** to specify that the authorize action should always be run before change. Note that authorize is declared **private** (line 59), so it cannot be called directly as an action.

When authorize is called, it looks up the product to be modified (line 60) 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 61). If so, then authorize evaluates to **nil** (line 61), and control passes to change, which updates the product description (line 53), commits the change to the database (line 54), and then calls info to show the product listing screen (line 55). Otherwise, authorize calls info (line 62), and since that ends in a **redirect_to**, the action change will never be rendered.

More Possible Errors Caught by DRails. Again, DRails can prevent several potential pitfalls in writing controllers and views.

- View file names could have the wrong extension, in which case Rails may be unable to find them, causing crashes or unintended behavior. A (perhaps implicit) call to **render** could go to a non-existent view. Furthermore, 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 (say, by writing :authorized instead of :authorize on line 44), or make a mistake in a **redirect_to** call (say, by writing "company" instead of "companies" on line 48, or @company = ... rather than company = ... on line 47). DRails catches such bugs by trying to explicitly insert the intended method calls and type checking the resulting code.
- Embedded code in views might make type errors when accessing fields (like @company) set in controllers. DRails checks for such errors plus other type errors in controller and view code.

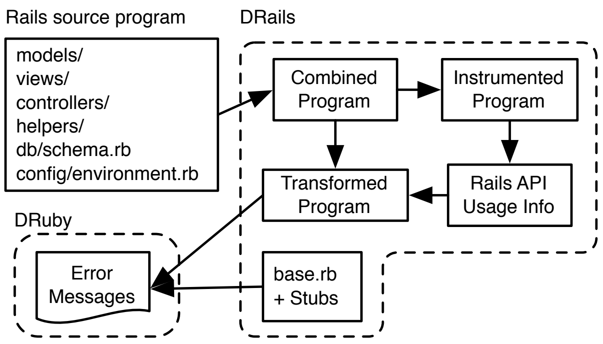


Figure 5. DRails architecture

Summing up, even an application as simple as *catalog* contains many opportunities for inadvertent mistakes, and Ruby's dynamic typing means that such errors can remaining latent until run time. In addition to the problems we have already seen, DRails can detect several other issues, such as type-incorrect calls to Rails API methods, using Rails features that are deprecated, and in general catching type errors in Ruby code. Next, we explore how DRails transforms Rails source code into Ruby, to allow us to use type inference to find these problems.

3. DRails: From Rails to Ruby

Figure 5 shows the basic architecture of DRails, which comprises approximately 1,700 lines of OCaml and 2,000 lines of Ruby. To run DRails, the user executes the command "drails *dirname*," where *dirname* is the root directory containing the Rails program. In addition to the application subdirectories we have already seen, Rails programs also include several other directories, parts of which are analyzed by DRails. The helpers/ directory contains Ruby code that may be shared across several models or controllers, and the file config/environment.rb has global configuration information such as external library imports and global constants.

As illustrated in Figure 5, DRails begins by combining all the separate files of the Rails application into one large program. 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 libraries used by the application. DRuby performs type inference and emits warnings for any errors it finds.

The most unusual feature of our approach is instrumenting the source code and then loading it into Ruby. Originally we used a purely static approach that found methods called and their argument values via pattern matching on the parsed Ruby source code. However, we found the pattern matching code to be ad hoc and tedious to write, since it needed to be specialized for all Rails API functions. Moreover, since in the dynamic approach we record method calls using a simple string-based encoding, it made it very easy to discover how a Rails application was calling the API, and DRuby uses a related dynamic analysis technique to good effect on regular Ruby code [5].

We next describe the steps that DRails uses to produce the various program representations in Figure 5.

Defining Models. Recall from the example in Figure 3 that Rails creates getter and setter methods in models based on the fields listed in db/schema.rb. This kind of low-level class manipulation is not typable in DRuby (or in any standard type system), so DRails makes the effect explicit by transforming the models to include getter and setter method definitions. For example, the Company model in Figure 3 is modified as follows:

```
class Company < ActiveRecord::Base
  attr_accessor :id, :name # inserted by DRails
  ...
```

The call to attr_accessor creates methods to read and write fields @id and @name.

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. (For an example, recall line 31 of Figure 4.) 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.

"Rubifying" Views. To fully reason about a Rails application, we need to be able to analyze the Ruby code embedded in views, and we wanted to do this without changing DRuby's rather complex parser. Our solution was to use Markaby for Rails [16] to parse the views and produce regular Ruby classes that generate the same web page. We call this process *Rubifying* the view. Note that while Markaby worked as-is initially on small examples, we needed to make major changes to apply it to our suite of programs in Section 4.

As an example, here is the result of Rubifying views/companies/info.html.erb of Figure 4, slightly simplified for discussion purposes:

```
69 module CompaniesView
70   include ActionView::Base
71   def info
72     Rubify.h2 do Rubify.text(@company.name) end
73     Rubify.h3 do Rubify.text("Products") end
74     Rubify.table do
75      @company.products.each do |product|
76        Rubify.tr do
77          Rubify.td do
78            Rubify.text(product.name)
79            Rubify.text("(")
80            Rubify.text(product.description)
81            Rubify.text(")")
82          end
83        end
84      end
85     end
86   end
87 end
```

Here we created a method info (based on the view name info). The calls to class methods of Rubify output strings containing the appropriate HTML text, and notice that the calls are intermixed with regular Ruby code. For example, line 72 creates the second-level heading on line 34 of Figure 4.

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. We make the view a **module** rather than a **class** for reasons we will discuss later in this section.

This step not only produces Ruby code we can analyze with DRuby, but DRails also does two other checks: It makes sure that the HTML is well-formed (in the sense that closing and opening tags are balanced), and it also ensures that the views' filename extensions match what Rails expects.

Combining a Rails Application and Gathering API Usage Data. DRails parses the application's source code (including the Rubified views) into the Ruby Intermediate Language (RIL), a subset of Ruby that is designed to be easy to analyze and transform [6]. RIL is DRuby's internal representation, and it can be unparsed into code that is semantically identical to the original source. DRails concatenates the RIL representation of each application component, creating a single, "combined" program that contains the whole application.

The next step is to discover what calls the program makes to the Rails API, so that we can make the effects of hard-to-statically analyze calls explicit in the source code. For example, in Section 2 we saw that calling **has_many** created methods to get and set database table relationships, and calling **before_filter** modified the sequencing of actions. To type programs that use these, then, DRails needs to add the actual method definitions and the implied calls to the program.

As mentioned earlier, we record information about these Rails API calls dynamically. We observed that essentially all of the calls we need to process are invoked as the model and controller source files are loaded. For example, the call to **before_filter** on line 44 in Figure 4 is actually invoked as the ProductsController class is loaded. Hence we use a "load time" analysis: At each API call of interest, we add instrumentation that records the location of the call in a global variable. We also created a file with mock definitions of **has_many**, **before_filter**, and all the other necessary Rails methods. Our mock functions simply record the method called, its arguments, and any additional information that is helpful in modeling the call. We then load the file with Ruby, which triggers the instrumentation calls, and the information we gather is stored in a data file that is then loaded by OCaml and used in the next step.

There are four groups of Rails API calls that DRails records: *filters*, such as **before_filter** and **after_filter**, which create chains of filters before and after actions; *associations*, such as **belongs_to** and **has_many**, which create methods to access database model relationships; *callbacks*, such as **validate**, which insert method calls whenever particular events happen (e.g., a model is saved to the database); and *layouts*, which specify a "template" view that is always invoked first in a controller and then calls out to other views.

Transforming Rails Programs for Analysis. Next, we use the Rails API call information to transform the original source program and make the behavior of the calls explicit. The particular transformation varies with the category of call.

Filters are eliminated but the appropriate calls are inserted in the controllers. For example, **before_filter** on line 44 of Figure 4 is removed, and the change method is modified to have an explicit call to authorize:

```
88    def change
89      authorize() # inserted by DRails
90      @product.description = params[:description]
91      @product.save
92      info
93    end
94    ...
```

Association calls are also removed, and the methods implied by them are added. One subtlety is that if there is a **has_many** relationship, then accessor methods return something that is actually a "monkey-patched" instance of Ruby's Array class. ("Monkey patching" means the object's methods are changed at run time.) We model this using a class HasManyCollection that we created to mimic the special return type of these methods. For example, the **has_many** call on line 11 of Figure 3 produces the following set of type annotations (and more):

```
95   class Company < ActiveRecord::Base
96     ##% products : () → HasManyCollection<Product>
97     ...
98     ##% products= : \
99     ##%   (Array<Product>) → HasManyCollection<Product>
100    ...
101  end
```

Line 96 annotates the getter method, which takes no arguments and returns an instance of HasManyCollection whose contents has type Product. Similarly, lines 98–99 annotate the setter method, which takes an Array and returns a HasManyCollection. More details on DRuby's type annotation language can be found elsewhere [6].

Callbacks are inserted into the appropriate positions in the code, similarly to filters. To illustrate some of the complexities, suppose we modified the Product model to also call **before_validation** :foo, which indicates method foo should be called before validation:

```
102  class Product < ActiveRecord::Base
103    belongs_to :company
104    validate :unique_name_in_company
105    before_validation :foo # added
106    ...
```

Then DRails transforms Product as follows:

```
107  class Company < ActiveRecord::Base
108    def validate()
109      before_validation() # inserted by DRails
110      unique_name_in_company()
111    end
112    def before_validation() # inserted by DRails
113      foo()
114    end
115    ...
```

Here DRails rewrote line 104 as the method on lines 108–111, and it rewrote line 105 as the method on lines 112–114. The key is that on line 109, our **validate** method calls the transformed code for **before_validation**. Then in base.rb, we define ActiveRecord::Base (which is inherited on line 107) so that save calls **validate**:

```
116  class ActiveRecord::Base
117    def save()
118      ...validate()...
119    end
120    def validate()
```

```
121     ...before_validation()...
122   end
123   ...
```

Notice that lines 120–122 also define a **validate** method, which is overridden in our transformed Company class. This lets us handle the case when **before_validation** is used with a non-custom validator (e.g., the call to **validates_uniqueness_of** on line 12 in Figure 3).

Lastly, layouts are modeled simply as regular view classes, except we ensure that if a layout is specified, it is always called first whenever a view is rendered.

Further Transformations. Our transformation phase also makes a few other changes. The most substantial is to support **render** and **redirect_to**. Recall from Section 2 that these methods invoke either views or actions according to their arguments. DRails makes these calls explicit so that DRuby can "see" them.

To do this, we modify the structure of controller and view classes in several ways. First, we duplicate each controller method—one copy stays as-is (in case it is called directly in the Ruby code; after all, it is an ordinary method), and the second copy is modified so the view it renders or controller it redirects to is called explicitly. For example, DRails modifies CompaniesController and CompaniesView as follows:

```
124 class CompaniesController < ApplicationController
125   include CompaniesView
126   def info
127     @company = Company.find_by_name(params[:name])
128   end
129   def __ctrl_info
130     @company = Company.find_by_name(params[:name])
131     __view_info()
132   end
133 end
134 module CompaniesView
135   include ActionView::Base
136   def __view_info
137     ...Rubify.h2 do Rubify.text(@company.name) end...
138   end
139 end
```

Notice that the copy of info on lines 126–128 is as before, but a duplicate copy of it on lines 129–132 has been made with name __ctrl_info. That version, instead of returning directly, ends with a call to __view_info(), the renamed version of the Rubified info method. Recall the view's method was called implicitly before.

Also notice that on line 137, the view is able to access a field set by the controller on line 130, even though they come from different classes. We model this in DRails by making CompaniesView a module (line 134) and then including it in CompaniesController (line 125). (We use a **module** because while a Ruby class can only have one superclass, it can inherit from many modules.) This inheritance is why we renamed the info method of the view to __view_info(), to avoid clashes with the info method of the controller.

Running DRuby. Finally, the last step is to apply DRuby to the transformed program. At this point, the Rails-specific analysis is complete, and we have replaced all the hard-to-analyze Rails API methods with equivalent code that we can check for type errors—and type errors in that code indicate problems in the original, untransformed Rails application.

We model the remainder of the Rails API with code and type annotations in base.rb. For example, here are signatures for two methods of ActiveRecord::Base, which is inherited by models:

```
140 module ActiveRecord
141   ...
142   class Base
143     ##% attributes<t> : () → Hash<Symbol, t>
144     ...
145     ##% attributes=<t> : \
146     ##%   (Hash<Symbol, t>, ?Boolean) → Hash<Symbol, t>
147     ...
```

The method attributes returns a hash mapping attribute names to their values. The method attributes= allows programmers to set multiple attributes at once by passing in a hash and, optionally, a boolean flag (indicating whether certain attributes may be changed by the call), and it returns the new attributes hash. In general, we give these API methods the most precise type signatures possible in DRuby. We should note that sometimes our type annotations are less precise than we would like, however, because some Rails API methods are extremely polymorphic or would require a dependent type system for full precision.

We include base.rb when we run DRuby on the transformed program. We also include stub files with annotations for portions of the Ruby standard library and other external libraries used by the Rails applications in our experiments.

4. Experiments

We evaluated DRails by running it on 11 Rails applications that we obtained from various sources including RubyForge, OpenSourceRails, and our colleagues. The first group of columns in Figure 6 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

		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

R - erb fix	H - directory reorganization	I - routing info	B - environment.rb
E - errors	W - warnings	D - deprecated	F - false positives

Figure 6. Experimental results

RIL control-flow graph after full transformation. Note that the last step of DRails's transformation increases the code size significantly, by a factor of 2.7 on average. This increase shows that there is a significant amount of code that Rails produces by convention.

We made four kinds of manual changes to applications to make them "DRails-compatible," summarized in the second group of columns in Figure 6. First, DRails cannot complete its translation if the .html.erb files in the application contain unbalanced tags, if tags are opened in HTML code and closed in Ruby code, or if embedded Ruby code contains syntax that DRuby cannot parse. We count the number of changes to correct these issues under (R).

Second, DRails requires that the directory structure of the application match the documented specification for Rails exactly, whereas Rails itself is slightly more forgiving. We needed to do minor reorganizations in the directory structure of *diamondlist* and *onyx*. We also needed to flatten some class names that had nested scope and move the class files accordingly. We count these cases under (H).

Third, sometimes **render** and **redirect_to** are called with non-constant arguments, or an application uses "RESTful routing" [20] instead, which DRails does not currently support. For these cases we manually specified the targets of **render** and **redirect_to**, and we count the number of times we needed to do this as (I).

Finally, since DRails does not automatically detect library imports, we had to add several require statements (which load another file) to config/environment.rb. In the same file, we also removed a call require "boot.rb", which loads the Rails framework, as this is unnecessary for DRails. These changes are listed as (B).

4.1. Results

The results of running DRails on these programs are tabulated in the last two groups of columns in Figure 6. We ran DRails on an AMD Athlon 4600 processor with 4GB of memory.

The second-to-last group of columns shows the running times of DRails. We break this down into the DRails-only time on the left, and the DRuby time in the middle; the total time is the sum of these two columns. The reported running times are the average of three runs. The DRails-only step is typically 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 correspond 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 that are no longer available in Rails 2.x; and *false positives* (F) that do not correspond to actual bugs. Recall from Section 3 that Rails duplicates code for actions in controllers. This may cause duplicate warnings, which we do not include in the counts.

Errors. 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.

As another example of this kind of error, *onyx* contains the code:

```
@count, @next, @last = 1
```

We contacted the developer and confirmed that he expected this to assign 1 to all three fields. However, this code only assigns 1 to @count, and sets @next and @last to **nil**. DRails catches this error as a type mismatch between Fixnum (the type of integers) and Array (the type expected at a parallel assignment in DRuby).

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.

Warnings. We found 2 warnings across our applications. The warning in *chuckslist* occurs in the code

```
@ad = Category.find(params[:category]).ads.new
```

Here ads returns a collection (a HasManyCollection in DRails). According to the Rails documentation, the programmer should therefore call create to make a new instance. However, although the new method is not mentioned in the documentation for this case, it appears to work. (This is a very confusing usage, because new is typically called only on instances of Class.)

The other warning occurs in *depot*, in which Hash's map method is used without an explicit tuple type for the block argument:

```
validates_inclusion_of :pay_type,
  :in => PAYMENT_TYPES.map {|disp, value| value}
```

The correct syntax for the block argument is |(disp,value)|, because map expects a single argument (a tuple) rather than two arguments. Ruby is fairly lenient in this particular case and pairs the two values before binding them to map's formal parameter. However, we consider this a bad programming practice because such pairing does not always happen automatically in Ruby [6].

Deprecated. We found 72 uses of deprecated constructs across five benchmarks. All of these cases cause run-time errors on Rails 2.x, though they operate correctly on older versions of Rails. Our applications often do not document what version of Rails they are intended to work with, so these may or may not be errors depending on the programmer's intention.

False positives. DRails reported 57 false positives. Twenty-nine of these (across eight benchmarks) are due to limitations in DRuby's annotation language; we sometimes had to assign overly general types to Rails API methods, and this could conflate types during inference and trigger false warnings.

Twenty-four of the false positives (across four benchmarks) are because DRails does not handle some Rails features, namely the ActionMailer, ActionController, and Configuration modules. We expect these could be addressed with more engineering effort.

Three false positives are due to DRails's Rubification step. Recall that DRails converts HTML tags to Ruby method calls with an optional block. The introduction of these block scopes means local variables in different blocks are different, but in the original view file they referred to the same variable. Again, this could be addressed with more engineering effort.

The last false positive is due to a run-time type test. DRuby does not realize that if the test passes, then the tested value has the given type. This could be solved by extending DRuby to include occurrence typing [24].

4.2. 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, as we saw in Section 3, our type signatures for Rails API methods are sometimes overly general, statically allowing calls that might fail at run time. Nevertheless, our experiments show that DRails is still quite useful in finding errors in Rails programs.

Second, DRails's modeling of the Rails API is incomplete and could be slightly inaccurate. Indeed, the Rails API is enormous, and many features of its API are poorly or not at all documented. In such cases, we had to examine Rails's source and use trial and error to understand the feature, and thus there could be mistakes in our interpretation. However, as we have applied DRails across a range of programs and produced sensible results, we believe we have correctly modeled the essential core of Rails.

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 run time. We addressed this concern by conferring among ourselves about the errors and getting the opinion of the developers for some of the problems we found.

Finally, there could be bugs in DRuby that cause it to unsoundly miss type errors. However, DRuby has been run on a significant amount of code at this point [6, 5], and so we believe any remaining unsoundness is likely minor.

5. Related Work

Static Analysis for Web Applications. Most existing work on static analysis of web applications focuses on verification of security properties. Lam et al. [13] combine static analysis with model checking to verify that information-flow patterns are satisfied in Java-like programs. Huang et al. [11] use a lattice-based static analysis algorithm derived from type systems and typestate to ensure similar information-flow properties. The tool TAJ [25] performs taint analysis of web applications written in Java, and uses novel program slicing techniques to handle reflective calls and flows through containers. On the other hand, the tool Pixy [12] performs alias analysis for PHP and finds security vulnerabilities in web applications written in PHP. Xie and Aiken [28] address the same problem, and present a static analysis algorithm based on symbolic evaluation to handle dynamic features of PHP. While the above papers focus on server-side code, Guha et al. [10] present a static control-flow analysis for client-side JavaScript code to handle dynamic code generation. Maffeis and Taly [15] study methods for filtering and rewriting JavaScript code to address similar problems.

The key differences between these systems and DRails is our focus on static typing and Ruby on Rails, a combination we believe we are the first to study.

Static Typing in Dynamically Typed Languages. Other related work focuses on the elimination of common programming mistakes in scripting languages through compiler support. WASH [22] is a Haskell-embedded language for server-side web scripting that provides extensive guarantees due to its pervasive use of type information. Thorn [27] is another scripting language, targeting the JVM, that allows optional constraint annotations to drive lightweight static type inference, and provides safety guarantees for the typed parts of a program while ensuring smooth integration with the untyped parts of the program [26]. Recently Gorbovitski et al. [8] implement an abstract interpretation framework for Python to provide a basis for program analysis. Maffeis et al. [14] provide an operational semantics for JavaScript to provide a similar basis for program analysis.

Finally, there is a lot of theoretical work on integrating static and dynamic typing in object-oriented languages, including gradual types [21] and hybrid types [3]. Some of these ideas have been implemented in (extensions of) JavaScript [2, 23], Python [1], Smalltalk [9], Scheme [24], and Ruby [6]. We build on top of the latter system, DRuby, in this work. A related but less powerful type inference algorithm appears in RadRails, an IDE for Rails [17, 18], to suggest methods during method completion in the IDE.

6. Conclusion

In this paper, we present 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 previously developed static type inference system for Ruby, to the result. We show that static typing can catch a variety of bugs in Rails applications that may cause exceptions or otherwise unintended behaviors at run time. We believe we are the first to bring static typing to Ruby on Rails.

There are several interesting directions for future work. We plan to continue extending DRails's analysis to more features of Rails, to increase its comprehensiveness. We also plan to consider ways to check correctness properties that are deeper than simple typing. Verifying such properties on Rails programs might require some interesting new techniques. Lastly, we intend to explore how far our approach can be applied to web applications written in related scripting languages such as Perl, Python, and PHP.

References

[1] D. Ancona, M. Ancona, A. Cuni, and N. Matsakis. RPython: Reconciling Dynamically and Statically Typed OO Languages. In *DLS*, 2007.

[2] C. Anderson, P. Giannini, and S. Drossopoulou. Towards Type Inference for JavaScript. In *ECOOP*, pages 428–452, 2005.

[3] C. Flanagan, S. N. Freund, and A. Tomb. Hybrid types, invariants, and refinements for imperative objects. In *FOOL*, 2006.

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

[5] M. Furr, J. An, and J. S. Foster. Profile-guided static typing for dynamic scripting languages. In *OOPSLA*, 2009. To appear.

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

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

[8] M. Gorbovitski, K. T. Tekle, and Y. A. Liu. Assessing alias analysis for object-oriented and dynamic languages, 2009. IBM PL Day Talk.

[9] J. O. Graver and R. E. Johnson. A type system for Smalltalk. In *PLDI*, pages 136–150, 1990.

[10] A. Guha, S. Krishnamurthi, and T. Jim. Using static analysis for Ajax intrusion detection. In *WWW*, 2009.

[11] 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*, pages 40–52, 2004.

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

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

[14] S. Maffeis, J. Mitchell, and A. Taly. An operational semantics for JavaScript. In *APLAS*, pages 307–325, 2008.

[15] S. Maffeis and A. Taly. Language-based isolation of untrusted Javascript. In *CSF*, 2009.

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

[17] J. Morrison. Type Inference in Ruby. Google Summer of Code Project, 2006.

[18] Radrails, 2008. http://www.aptana.com/rails.

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

[20] S. Ruby, D. Thomas, and D. H. Hansson. *Agile Web Development with Rails, Third Edition*. The Pragmatic Bookshelf, 2009.

[21] J. Siek and W. Taha. Gradual typing for objects. In *ECOOP*, pages 2–27, 2007.

[22] P. Thiemann. An embedded domain-specific language for type-safe server-side web-scripting. *ACM Transactions on Internet Technology*, 2003.

[23] P. Thiemann. Towards a type system for analyzing javascript programs. In *ESOP*, pages 408–422, 2005.

[24] S. Tobin-Hochstadt and M. Felleisen. The Design and Implementation of Typed Scheme. In *POPL*, pages 395–406, 2008.

[25] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. TAJ: Effective taint analysis for Java. In *PLDI*, 2009. To appear.

[26] T. Wrigstad, F. Z. Nardelli, S. Lebresne, J. Östlund, and J. Vitek. Integration of typed and untyped code in Thorn, 2009. Submitted.

[27] T. Wrigstad, J. Östlund, G. Richards, J. Vitek, B. Bloom, J. Field, N. Nystrom, and R. Strnisa. Thorn—Robust, concurrent, extensible scripting on the JVM. In *OOPSLA*, 2009. To appear.

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