Static Type Inference for Ruby

Michael Furr   Jong-hoon (David) An   Jeffrey S. Foster   Michael Hicks

Department of Computer Science
University of Maryland
College Park, MD 20742
{furr,davidan,jfoster,mwh}@cs.umd.edu

ABSTRACT

Many general-purpose, object-oriented scripting languages are dynamically typed, which provides flexibility but leaves the programmer without the benefits of static typing, including early error detection and the documentation provided by type annotations. This paper describes Diamondback Ruby (DRuby), a tool that blends Ruby's dynamic type system with a static typing discipline. DRuby provides a type language that is rich enough to precisely type Ruby code we have encountered, without unneeded complexity. When possible, DRuby infers static types to discover type errors in Ruby programs. When necessary, the programmer can provide DRuby with annotations that assign static types to dynamic code. These annotations are checked at run time, isolating type errors to unverified code. We applied DRuby to a suite of benchmarks and found several bugs that would cause run-time type errors. DRuby also reported a number of warnings that reveal questionable programming practices in the benchmarks. We believe that DRuby takes a major step toward bringing the benefits of combined static and dynamic typing to Ruby and other object-oriented languages.

Categories and Subject Descriptors

F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—Program analysis

General Terms

Languages, Verification

Keywords

dynamic typing, Ruby, type inference, contracts

1. INTRODUCTION

Dynamic type systems are popular in general-purpose, object-oriented scripting languages like Ruby, Python and Perl. Dynamic typing is appealing because it ensures that no correct program execution is stopped prematurely—only programs about to "go wrong" at run time are rejected.

However, this flexibility comes at a price. Programming mistakes that would be caught by static typing, e.g., calling a method with the wrong argument types, remain latent until run time. Such errors can be painful to track down, especially in larger programs. Moreover, with pure dynamic typing, programmers lose the concise, automatically-checked documentation provided by type annotations. For example, the Ruby standard library includes textual descriptions of types, but they are not used by the Ruby interpreter in any way, and in fact, we found several mistakes in these ad hoc type signatures in the process of performing this research.

To address this situation, we have developed Diamondback Ruby (DRuby), an extension to Ruby that blends the benefits of static and dynamic typing. Our aim is to add a typing discipline that is simple for programmers to use, flexible enough to handle common idioms, that provides programmers with additional checking where they want it, and reverts to run-time checks where necessary. DRuby is focused on Ruby, but we expect the advances we make to apply to many other scripting languages as well. Our vision of DRuby was inspired by an exciting body of recent work on mixing static and dynamic typing, including ideas such as soft typing [14], the dynamic type [2], gradual typing [33], and contracts [16].

Designing DRuby's type system has been a careful balancing act. On the one hand, we would like to statically discover all programs that produce a type error at run time. On the other hand, we should not falsely reject too many correct programs, lest programmers find static typing too restrictive. Our approach is to compromise on both of these fronts: We accept some programs that are dynamically incorrect, and reject some programs that are dynamically correct.

In particular, we use type inference to model most of Ruby's idioms as precisely as possible without any need for programmer intervention. For example, we track the types of local variables flow-sensitively through the program, e.g., allowing a variable to first contain a String and then later an Array. On the other hand, we provide no precise characterization of Ruby's more dynamic features, such as metaprogramming with eval or removing methods with Module.remove_method. Incorrect usage of these features could lead to a run-time type error that DRuby fails to warn about.

Between these two extremes lies some middle ground: DRuby might not be able to infer a static type for a method, but it may nonetheless have one. For example, DRuby supports but does not infer intersection and universal types. To model these kinds of types, DRuby provides an annotation language that allows programmers to annotate code with types that are assumed correct at compile time and then are checked dynamically at run time. While the Ruby interpreter already safely aborts an execution after a run-time type error, our checked annotations localize errors and properly *blame* [16] the errant code.

To summarize, DRuby makes three main contributions. First,

DRuby includes a type language with features we found necessary to precisely type Ruby code: union and intersection types [28], object types (to complement nominal types) [1], a self type [12], parametric polymorphism [29], tuple types for heterogeneous arrays, and optional and variable arguments in method types.

Second, DRuby includes a surface type annotation syntax. Annotations are required to give types to Ruby's core standard library, since it is written in C rather than Ruby, and they are also useful for providing types for Ruby code that uses hard-to-analyze features. These annotations are dynamically checked and provide blame tracking, i.e., statically typed code is never blamed for a run-time type error. We believe our annotation language is easy to understand, and indeed it resembles the "types" written down informally in the standard library documentation.

Finally, DRuby includes a type inference algorithm to statically discover type errors in Ruby programs, which can help programmers detect problems much earlier than dynamic typing. By providing type inference, DRuby helps maintain the lightweight feel of Ruby, since programmers need not write down extensive type annotations to gain the benefits of static typing.

We have applied DRuby to a suite of benchmark programs ranging from 29–1030 lines of code. Our benchmarks are representative of the kinds of scripts programmers write with Ruby. DRuby found these 18 benchmarks to be largely amenable to static typing: it reported 5 potential errors and 16 warnings for questionable code compared to 16 false positives.

We believe that DRuby takes a major step toward combining the benefits of static and dynamic typing in object-oriented programming languages.

2. STATIC TYPES FOR RUBY

We present DRuby's type system by example; we elide a full formal presentation due to space constraints. The design of DRuby was driven by experience—we included features expressive enough to type idioms common in our benchmark programs and the standard library APIs, but at the same time we tried to keep types easy for programmers to understand.

Basic Types and Type Annotations. In Ruby, everything is an object, and all objects are class instances. For example, 42 is an instance of Fixnum, **true** is an instance of TrueClass, and an instance of a class defined with **class** A ... **end** is created with A.new. Thus, a basic DRuby type is simply a class name, e.g., Fixnum, TrueClass, A, etc. Classes can extend other classes, but a subclass need not be a subtype. The class Object is the root of the class hierarchy.

Although Ruby gives the illusion that built-in values such as 42 and **true** are objects, in fact they are implemented inside the Ruby interpreter in C code, and thus DRuby cannot infer the types of these classes. However, we can declare their types using DRuby's type annotation language. In DRuby, type annotations appear before the corresponding class or method declaration. All annotations appear on a line beginning with ##%, and therefore appear as comments to the standard Ruby interpreter.

We developed a file `base_types.rb` that annotates the "core library," which contains classes and globals that are pre-loaded by the Ruby interpreter and are implemented purely in C. This file includes types for portions of 185 classes and 17 modules, using 997 lines of type annotations in total. Our implementation analyzes `base_types.rb` before applying type inference to a program.

For example, here is part of the declaration of class String, with ##% removed for clarity:

```
class String
    "+" : (String) → String
```

```
    insert  : (Fixnum, String) → String
    ...
end
```

The first declaration types the method + (non-alphanumeric method names appear in quotes), which concatenates a String argument with the receiver and returns a new String. Similarly, the next line declares that insert takes a Fixnum (the index to insert at) and another String, and produces a new String as a result.

Intersection Types. Many methods in the standard library have different behaviors depending on the number and types of their arguments. For example, here is the type of String's include? method, which either takes a Fixnum representing a character and returns true if the object contains that character, or takes a String and performs a substring test:

```
include? : (Fixnum) → Boolean
include? : (String) → Boolean
```

The type of include? is an example of an *intersection type* [28]. A general intersection type has the form t **and** t', and a value of such a type has *both* type t and type t'. For example, if A and B are classes, an object of type A **and** B must be a common subtype of both A and B. In our annotation syntax for methods, the **and** keyword is omitted (only method types may appear in an intersection), and each conjunct of the intersection is listed on its own line.

Another example of intersection types is String's slice method, which returns either a character or a substring:

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

Notice that this type has quite a few cases, and in fact, the Ruby standard library documentation for this function has essentially the same type list.[1] Intersection types serve a purpose similar to method overloading in Java, although they are resolved at run time via type introspection rather than at compile time via type checking. Annotation support for intersection types is critical for accurately modeling key parts of the core library—74 methods in `base_types.rb` use intersection types. Note that our inference system is currently unable to infer intersection types, as method bodies may perform ad-hoc type tests to differentiate various cases, and so they can currently be created only via annotations.

Optional Arguments and Varargs. One particularly common use of intersection types is methods with optional arguments. For example, String's chomp method has the following type:

```
chomp : () → String
chomp : (String) → String
```

Calling chomp with an argument s removes s from the end of self, and calling chomp with no arguments removes the value of (global variable) $/ from self. Since optional arguments are so common, DRuby allows them to be concisely specified by prefixing an argument type with ?. The following type for chomp is equivalent to the intersection type above:

```
chomp : (?String) → String
```

DRuby also supports varargs parameters, specified as $*t$, meaning zero or more parameters of type t (the corresponding formal argument would contain an Array of t's). For example, here is the type of delete, which removes any characters in the intersection of its (one or more) String arguments from self:

[1] http://ruby-doc.org/core/classes/String.html#M000858

$$\text{delete} : (\text{String}, *\text{String}) \rightarrow \text{String}$$

Notice this type is equivalent to an intersection of an unbounded number of types.

Union Types. Dually to intersection types, DRuby supports *union types*, which allow programmers to mix different classes that share common methods. For example, consider the following code:

```
class A; def f () end end
class B; def f () end end
x = ( if  ...  then A.new else B.new)
x.f
```

Even though we cannot statically decide if x is an A or a B, this program is clearly well-typed at run time, since both classes have an f method. Notice that if we wanted to write a program like this in Java, we would need to create some interface I with method f, and have both A and B implement I. In contrast, DRuby supports union types of the form t **or** t', where t and t' are types (which can themselves be unions) [24]. For example, x above would have type A **or** B, and we can invoke any method on x that is common to A and B. We should emphasize the difference with intersection types here: A value of type A **and** B is both an A and a B, and so it has *both* A's and B's methods, rather than the union type, which has one set of methods or the other set, and we do not know which.

Note that the type Boolean used in the type of include? above is equivalent to TrueClass **or** FalseClass (the classes of true and false, respectively). In practice we just treat Boolean as a pseudo-class, since distinguishing true from false statically is essentially useless—most uses of Booleans could yield either truth value.

The self type. Consider the following code snippet:

```
class A; def me() self end end
class B < A; end
B.new.me
```

Here class A defines a method me that returns **self**. We could naively give me the type () $\rightarrow$ A, but observe that B inherits me, and the method invocation on the last line returns an instance of B, not of A. Thus we include the type of self as an implicit parameter to every method and bind it to the receiver. Similarly, the clone method of Kernel (a module included in all other classes) has type () $\rightarrow$ **self**. Self annotations are de-sugared into a form of parametric polymorphism, described below.

Object Types. Thus far we have only discussed types constructed from class names and self. However, we need richer types for inference, so we can describe objects whose classes we do not yet know. For example, consider the following code snippet:

```
def f(x)  y = x.foo; z = x.bar;  end
```

If we wanted to stick to nominal typing only, we could try to find all classes that have methods foo and bar, and then give x a type that is the union of all such classes. However, this would be both extremely messy and non-modular, since changing the set of classes might require updating the type of f.

Instead, DRuby includes *object types* $[m_0 : t_0, \ldots, m_n : t_n]$, which describes an object in which each method m_i has type t_i. The parameter x above has type [foo : () $\rightarrow$ t, bar : () $\rightarrow$ u] for some t and u. As another example, `base_types.rb` gives the print method of Kernel the type

$$\text{print} : (*[\text{to\_s} : () \rightarrow \text{String}]) \rightarrow \text{NilClass}$$

Thus print takes zero or more objects as arguments, each of which has a no-argument to_s method that produces a String. Object types are critical to describe user code, but they are not that common in annotations for the core library. Only eight methods in `base_types.rb` include object types.

Note that we do not include fields in the annotation syntax for object types since they cannot be accessed outside of the class's methods. During inference, we model fields flow-insensitively, giving a field the same type across all instances of its class.

Parametric Polymorphism. To give precise types to container classes, we use *parametric polymorphism*, also called *generics* in Java. For example, here is part of the Array class type, which is parameterized by a *type variable* t, the type of the array contents:

```
class Array<t>
  at : (Fixnum) → t
  collect <u> : () {t → u} → Array<u>
  ...
end
```

As usual, type variables bound at the top of a class can be used anywhere inside that class. For example, the at method takes an index and returns the element at that index. Methods may also be parametrically polymorphic. For example, for any type u, the collect (map) method takes a code block (higher-order function) from t to u and produces an array of u.

DRuby uses parametric polymorphism internally for self types by including the method receiver as an implicit argument in the method signature. For example, we translate the method type for clone to the type <u> : (u) $\rightarrow$ u, where the receiver object of the method call is passed as the first argument.

Mixins. Ruby includes support for *modules* (a.k.a *mixins* [11]) for implementing multiple inheritance. For example, the following code defines a module Ordered, which includes an leq method that calls another method $\Leftrightarrow$ (three-way comparison). Class Container mixes in Ordered via include and defines the needed $\Leftrightarrow$ method.

```
module Ordered            # module/mixin creation
  def leq(x)
    (self ⇔ x) ≤ 0        # note "⇔" does not exist here
end end

class Container
  include Ordered          # mix in module
  def ⇔(other)             # define required method
    @x ⇔ other.get
end end
```

Standard object-oriented type systems would check all the methods in a class together, but DRuby cannot do that because of mixins, e.g., when typing leq, the actual $\Leftrightarrow$ method referred to depends on where Ordered is included. Instead, whenever a method m invokes another method on self, DRuby adds the appropriate constraints to self, e.g., when typing leq, DRuby adds a constraint that self has a $\Leftrightarrow$ method. Then when m is called, we check those constraints against the actual receiver. Typically modules also have polymorphic type signatures, so that they can be mixed into several different classes without conflating their types.

Tuple Types. The type Array<t> describes *homogeneous* arrays in which each element has the same type. However, since it is dynamically typed, Ruby also allows programmers to create heterogeneous arrays, in which each element may have a different type. This is especially common for returning multiple values from a function, and there is even special parallel assignment syntax for it. For example, the following code

```
def f () [1, true] end
a, b = f
```

assigns 1 to a and true to b. If we were to type f's return value as a homogeneous Array, the best we could do is Array<Fixnum **or** Boolean>, with a corresponding loss of precision.

DRuby includes a special type Tuple<$t_1, \ldots, t_n$> that represents an array whose element types are, left to right, t_1 through t_n. When we access an element of a Tuple using parallel assignment, we then know precisely the element type.

Of course, we may initially decide something is a Tuple, but then subsequently perform an operation that loses the individual element types, such as mutating a random element or appending an Array. In these cases, we apply a special subsumption rule that replaces the type Tuple<$t_1, \ldots, t_n$> with the type Array<t_1 **or** . . . **or** t_n>.

First Class Methods. DRuby includes support for another special kind of array: method parameter lists. Ruby's syntax permits at most one code block (higher-order function) to be passed to a method. For example, we can map $\lambda x.x+1$ over an array as follows:

```
[1, 2, 3]. collect {|x| x + 1}    # returns [2, 3, 4]
```

If we want to pass multiple code blocks into a method or do other higher-order programming (e.g., store a code block in a data structure), we need to convert it to a Proc object:

```
f = Proc.new {|x| x + 1}    # f is λx.x+1
f. call (3)                 # returns 4
```

A Proc object can be constructed from any code block and may be called with any number of arguments. To support this special behavior, in `base_types.rb`, we declare Proc as follows:

```
class Proc<ˆargs,ret>
    initialize  : () {(ˆargs) → ret} → Proc<ˆargs,ret>
    call  : (ˆargs) → ret
end
```

The Proc class is parameterized by a parameter list type ˆargs and a return type ret. The ˆ character acts as a type constructor allowing parameter list types to appear as first class types. The initialize method (the constructor called when Proc.new is invoked) takes a block with parameter types ˆargs and a return type ret and returns a corresponding Proc. The call method takes then has the same parameter and return types.

As another example use of ˆ, consider the Hash class:

```
class Hash<k, v>
    initialize  : () {(Hash<k, v>, k) → v} → Hash<k, v>
    default_proc  : () → Proc<ˆ(Hash<k, v>, k),v>
end
```

The Hash class is parameterized by k and v, the types of the hash keys and values, respectively. When creating a hash table, the programmer may supply a default function that is called when an accessing a non-existent key. Thus the type of initialize includes a block that is passed the hash table (Hash<k,v>) and the missing key (k), and produces a value of type v. The programmer can later extract this method using the default_proc method, which returns a Proc object with the same type as the block.

Types for Variables and Nil. DRuby tracks the types of local variables flow-sensitively, maintaining a per-program point mapping from locals to types, and combining types at join points with unions. This allows us to warn about accesses to locals prior to initialization, and to allow the types of local variables to vary from one statement to another. For example, in the following code

```
b = 42     # b is a Fixnum (no length method)
b = "foo"  # b is now a String (has a length method)
b.length   # only look for length in String, not Fixnum
```

we need flow-sensitivity for locals so to permit the call b.length.

We have to be careful about tracking local variables that may be captured by blocks. For example, in the code

```
x = 1
foo () { |y| x = y }   # pass in function λy.x=y
```

the value of x in the outer scope will be changed if foo invokes the code block. To keep our analysis simple, we track potentially-captured variables flow-insensitively, meaning they have the same type throughout the program. We also model class, instance (fields), and global variables flow-insensitively.

Finally, as it is common in statically typed OO languages, we treat nil as if it is an instance of any class. Not doing so would likely produce an excessive number of false alarms, or require a very sophisticated analysis.

Constants and Scoping. Class names in Ruby are actually just constants bound to special class objects. In general, constants, which are distinguished by being capitalized, can hold any value, and may be nested inside of classes and modules to create namespaces. For instance,

```
class A
  class B
    X = 1
end end
```

defines the classes A and A::B, and the non-class constant A::B::X. Identifying the binding of a particular constant is actually rather tricky, because it involves a search in the lexical scope in which the constant was created as well as the superclasses and mixins of the enclosing class. DRuby attempts to resolve the namespaces of all constants statically, and uses this information to construct the class hierarchy. For instance, in the definition **class** A < B, we need to find the binding for B to give A the right type.

Unsupported features. As we stated in the introduction, DRuby aims to be flexible enough to type common Ruby programming idioms without introducing needless complexity in its type system. Thus, there are a number of uses of Ruby that DRuby cannot type.

First, there are standard library methods with types DRuby cannot represent. For example, there is no finite intersection type that can describe Array.flatten, which converts an n-dimensional array to a one-dimensional array for any n.

Second, some features of Ruby are difficult for any static type system. In particular, Ruby allows classes and methods to be changed arbitrarily at run time (e.g., added to via class reopening or removed via the special undef statement). To keep DRuby practical, we assume that all classes and methods defined somewhere in the code base are available at all times. This may cause us to miss some type errors, and we leave addressing this assumption to future work.

Third, in Ruby, each object has a special *eigenclass* that can be modified without affecting other objects. For example, suppose x and y are both instances of Object. Then **def** x.foo () ... **end** adds method foo to the eigenclass of x but leaves y unchanged. Thus, after this declaration, we can invoke x.foo but not y.foo. DRuby is unable to model eigenclasses because it cannot always decide statically which object's type is being modified.

Finally, Ruby includes reflection (e.g., accessing fields by calling instance_variable_get) and dynamic evaluation (the eval method). Combined with the ability to change classes at run time, this gives programmers powerful metaprogramming tools. For example, our *text-highlight* benchmark includes the following:

```
ATTRIBUTES.each do |attr|
    code = "def #{attr}(&blk) ... end"
    eval code
  end
```

This code iterates through ATTRIBUTES, an array of strings. For each element it creates a string code containing a new method definition, and then evaluates code. The result is certainly elegant—methods are generated dynamically based on the contents of an array. However, no reasonable static type system will be able to analyze this code. Our analysis has no special knowledge of these constructs, and so when we encounter this code, we simply check that each is passed arguments of the correct type and ignore its evaluation behavior entirely (and would therefore emit a false positive if these methods are called). Ultimately, we think that approaches that mix analysis at compile time with analysis at run time (as done by RPython [7]) may be the best option.

3. TYPE INFERENCE

Our type inference system is constructed as a *constraint-based analysis*. We first traverse the entire program (including base_types.rb), visiting each statement once and generating constraints that capture dependencies between types. For example, if we see x.m(), we require that x have method m() by generating a constraint $typ_x \leq [m : () \rightarrow \alpha]$, meaning the type of x is a subtype of an object type with an m method. In general, constraints have the form $\kappa \leq \kappa$ where κ ranges over various kinds of types (objects, parameter lists, blocks, etc.).

We resolve the generated set of constraints by exhaustively applying a set of rewrite rules. For example, given $typ \leq \alpha$ and $\alpha \leq typ'$, we add the closure constraint $typ \leq typ'$. During this process, we issue a warning if any inconsistencies arise. For example, given $A \leq [m : meth\_typ]$, if the class A has no method m, we have found a type error. If we detect no errors, the constraints are satisfiable, and we have found a valid typing for the program.

Our constraint resolution process is a variation on fairly standard techniques [5, 19]. For example, given a constraint $A \leq [m : meth\_typ]$, where A is a class name, we derive a new constraint $A(m) \leq meth\_typ$, where $A(m)$ looks up m in A, searching for m in the superclasses and mixins of A if needed. When working with tuple types $(typ_1, \ldots, typ_n)$ we allow them to be treated as array types, as outlined in Section 2.

When solving a constraint with unions on the right hand side, e.g., $typ_1 \leq typ_2$ or typ_3, we require typ_1 to have a fully-resolved type that is equal to (not a subtype of) either typ_2 or typ_3. These restrictions mean DRuby cannot find a solution to all satisfiable constraint systems, but keeps solving tractable. We place a similar restriction on constraints with intersection on the left-hand side.

4. CAST INSERTION

Annotations allow DRuby to interface with code it cannot statically verify, either because the code is written in C, or because it is too expressive for our type system. However, because DRuby trusts annotations to be correct, improperly annotated code may cause run-time type errors, and these errors may be misleading. For example, consider the following code:

```
1    ##% evil : () → Fixnum
2    def evil () eval ("2. to_s ()' ') end
3    def f () return( evil () ∗ 3) end
4    f()−4
```

Here we have annotated the evil method on line 1 to return a Fixnum, and given this assumption, DRuby verifies that the remainder of the program is statically type safe. However, evil uses eval to return the string "2." This does not cause a type error on line 3 because String has a method ∗ : Fixnum→String, but then the resulting String is returned to line 4, where the Ruby interpreter finally raises a NoMethodError, since String does not have a − method.

There are two problems here. First, DRuby has certified that line 4 will never cause a type error, and yet that is exactly what has occurred. Second, the error message gives little help in determining the source of the problem, since it accuses the return value of f of being incorrect, rather than the return value of evil.

We can solve this problem with higher-order contracts [16], which allow us to attribute the *blame* for this example correctly. To ensure that a method matches its annotation, we instrument each annotated method with run-time checks that inspect the parameters and return values of the method. DRuby ensures that "inputs" have the correct type, and the dynamic checks ensure the "outputs" match the static types provided to DRuby.

There is one catch, however: if a method has an intersection type, we may not know statically which parameter types were passed in, which might affect what result type to check for. For example, consider the following method:

$$foo : (String) \rightarrow String$$
$$foo : (Fixnum) \rightarrow Fixnum$$

Whether the result should be a String or a Fixnum depends on the input parameter type. Our instrumented code for this example (slightly simplified) looks like the following:

```
1    alias untyped_foo foo
2    def foo(arg)
3      sigs = [[ String , String ],[ Fixnum,Fixnum]]
4      result = untyped_foo(arg)
5      fail () unless sigs.include? [arg.class,result .class]
6    end
```

On line 1, we save the old foo method under the name untyped_foo before defining a new version of foo on line 2. The annotated signature is then stored as a Ruby value in the variable sigs (line 3), and the original method is called on line 4. Finally, on line 5 we check to ensure that the combination of the argument class and return class are included in the list of possible signatures.

Currently, we support run-time checking for nominal types, union and intersection types, parameter lists with regular, optional, and vararg arguments, blocks, and polymorphic methods. We plan on expanding our support for annotations in future work, such as run-time checks for object types, annotations on individual expressions, and statically checked annotations.

5. IMPLEMENTATION

Ruby is a dauntingly large language, with many constructs and complex control flow. For instance, most branching statements have multiple forms, such as **if** e1 **then** e2 or the equivalent e2 **if** e1. Other statements like **return** e have behavior that varies by context. For instance, an occurrence of **return** e inside a block may cause control to return from the current block or from the enclosing method, depending on the form of the block definition.

To analyze Ruby programs, we developed a GLR parser for Ruby that produces an AST that preserves the details of the surface syntax. We then translate this AST into a smaller, simpler intermediate language we call the Ruby Intermediate Language (RIL). RIL normalizes redundant constructs, separates side-effect free expressions from statements, and makes all control flow explicit. Combined, these transformations make writing static analyses with RIL much easier than working directly with Ruby source code. DRuby maintains a mapping between RIL code and positions in the original Ruby source so we can report errors in terms of the Ruby code. Together, DRuby and RIL comprise approximately 15,000 lines of OCaml and 1,900 lines for the lexer and parser.

DRuby is a drop-in replacement for Ruby: the programmer invokes "druby *filename*" instead of "ruby *filename*." DRuby also

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

Figure 1: Experimental Results

accepts several custom options to control its behavior, e.g., to specify the location of `base_types.rb`. Another option instructs DRuby to execute the script with Ruby after performing its analysis (even in the presence of static type errors), which can be useful for testing specific execution paths during development. When DRuby is run, it first loads in the library annotations in `base_types.rb`, and then analyzes the "main" program passed as a command line argument. Ruby programs can load code from other files by invoking either `require` or `load`. When DRuby sees a call to one of these methods, it analyzes the corresponding file if the name is given by a string literal, and otherwise DRuby issues an error.

Language constructs like `new` and `include`, which appear to be primitives, are actually method calls in Ruby. However, because they implement fundamental language operations, DRuby recognizes calls to these methods syntactically and models them specially, e.g., creating a new class instance, or adding a mixin module. Thus if a Ruby program redefines some of these methods (a powerful metaprogramming technique), DRuby may report incorrect results. DRuby also has special handling for the methods `attr`, `attr_accessor`, `attr_writer`, and `attr_reader`, which create getter and/or setter methods named according to their argument.

6. EXPERIMENTAL EVALUATION

We evaluated DRuby by applying it to a suite of programs gathered from our colleagues and RubyForge. We believe these programs to be representative of the kind of small to medium sized scripts that people write in Ruby and that can benefit from the advantages of static typing. The left portion of Figure 1 lists the benchmark names, their sizes as computed by SLOCCount [40], and the number and kinds of changes required to be analyzable by DRuby (discussed below). The analyzed code includes both the application or library and any accompanying test suite.

Our current implementation does not analyze the standard library due to its frequent use of dynamic language features. Instead, we created a stub file with type annotations for the portion of the standard library, 120 methods in total, used by our benchmarks. Surprisingly, we found that although the standard library itself is quite dynamic, the APIs revealed to the user very often have precise static types in DRuby. For those benchmarks that supplied test suites (8 of the 18), we ran the test suite with and without dynamic annotation checking enabled. None of the dynamic checks failed, and the overhead was minimal. Five test suites took less than one

second with and without dynamic checking. The remaining three test suites ran in 2, 6, and 53 seconds and had overheads of 7.7%, 0.37%, and -12% respectively. The last test suite actually ran faster with our instrumentation, likely due to interactions with the Ruby virtual machine's method cache.

We made three kinds of changes to the benchmarks to analyze them. First, the benchmarks labeled with "R" included `require` calls that were passed dynamically constructed file names. For instance, the test suite in the *StreetAddress* benchmark contained the code

```
require File.dirname(__FILE__) + ' ../ lib /street_address'
```

In the benchmarks that used this technique, we manually replaced the argument of `require` with a constant string.

Second, several of the benchmarks used a common unit testing framework provided by the standard library. This framework takes a directory as input, and invokes any methods prefixed by `test_` contained in the Ruby files in the directory. To ensure we analyzed all of the testing code, we wrote a stub file that manually required and executed these test methods, simulating the testing harness. Benchmarks with this stub file are labeled with "S."

Finally, the *text-highlight* benchmark used metaprogramming (as shown in Section 2) to dynamically generate methods in two places. DRuby did not analyze the generated methods and hence thought they were missing, resulting in 302 false positives the first time we ran it. We manually replaced the metaprogramming code with direct method creation, eliminating this source of imprecision.

6.1 Experimental Results

The right portion of Figure 1 shows the running times for DRuby, the number of errors, the number of warnings, and the number of false positives. Times were the average of 5 runs on an AMD Athlon 4600 processor with 4GB of memory. We define *errors* (E) as source locations that may cause a run-time type error, given appropriate input, and *warnings* (W) as locations that do not cause a run-time error for any input, but involve questionable programming practice. *False positives* (FP) are locations where DRuby falsely believes a run-time error can occur, but the code is actually correct.

For these 18 programs, DRuby runs quickly (under 7 seconds except four mid-size benchmarks), producing 37 messages that we classified into 5 errors, 16 warnings, and 16 false positives.

Errors. The 5 errors discovered by DRuby occurred in 4 benchmarks. The error in *ai4r* was due to the following code:

```
return rule_not_found if !@values.include?(value)
```

There is no `rule_not_found` variable in scope at this point, and so if the condition ever returns true, Ruby will signal an error. Interestingly, the *ai4r* program includes a unit test suite, but it did not catch this error because it did not exercise this branch. The two errors in *vimrecover* were also due to undefined variables, and both occurred in error recovery code.

The error in *hashslice* is interesting because it shows a consequence of Ruby's expressive language syntax, which has many subtle disambiguation rules that are hard to predict. In *hashslice*, the test suite uses assertions like the following:

```
assert_nothing_raised { @hash['a', 'b'] = 3, 4 }
assert_kind_of (Fixnum, @hash['a', 'b'] = 3, 4)
```

The first call passes a code block whose body calls the []= method of @hash with the argument [3,4]. The second call aims to do a similar operation, verifying the return value is a Fixnum. However, the Ruby interpreter (and DRuby) parse the second method call as having three arguments, not two as the author intended (the literal 4 is passed as the third argument). In the corresponding standard library stub file, this method is annotated as the following:

assert_kind_of<t> : (Class, t, ?String) → NilClass

Because 4 is a Fixnum and not a String, DRuby considers this a type error. Although our type annotation matches the documentation for assert_kind_of, the actual implementation of the method coerces its third argument to a string, thus masking this subtle error at run time.

The last error is in *ObjectGraph*, which contains the following:

```
$baseClass = ObjectSpace.each_object(Class)
  { |k| break k if k.name == baseClassName }
```

The method each_object takes a code block, applies it to each element of the collection, and returns the total number of elements visited (a Fixnum). However, the block supplied above terminates the iteration early (**break** k), returning a specific element that has type Class. The programmer intended the loop to always terminate in this fashion, using $baseClass throughout the program as a Class, not a Fixnum. However, it is easy to make the loop terminate normally and therefore return a Fixnum, since the above code depends on data provided by the end user.

Warnings. 14 of the 16 reported warnings are due to a similar problem. Consider the code "5.times { |i| print "*" }", which prints five stars: The times function calls the code block n times (where n is the value of the receiver), passing the values $0..n-1$ as arguments. In the above code, we never used parameter i, but we still included it. However, Ruby allows blocks to be called with too many arguments (extra arguments are ignored) or too few (missing arguments are set to **nil**). We think this is bad practice, and so DRuby reports an error if blocks are called with the wrong number of arguments, resulting in these 14 warnings. If the user intends to ignore block parameters, they can supply an anonymous vararg parameter such as 5.times {|∗| ...} to suppress the warning.

Of the two remaining warnings, one occurs in *substitution_solver*, which contains the block arguments |tetragraph, freq| instead of |(tetragraph, freq)|. In the former case, DRuby interprets the block as taking two arguments, whereas the latter is a single (tuple) argument. As each calls the block with only a single argument, DRuby signals an error. As it turns out, Ruby is fairly lenient, and allows the programmer to omit the parentheses in this case. However, we feel this leads to confusing code, and so DRuby always requires parentheses around tuples used in pattern matching.

The last warning occurs in *relative*, which, similarly to *ObjectGraph*, calls an iterator where all executions are intended to exit via a **break** statement. In this case, the normal return path of the iterator block appears to be infeasible, and so we classify this as a warning rather than an error.

False positives. DRuby produced a total of 16 false positives, due to several causes. Three false positives are due to union types that are discriminated by run-time tests. For example, one of the methods in the *sudokusolver* benchmark returns either **false** or an array, and the clients of this method check the return values against **false** before using them as arrays. DRuby does not model type tests to discriminate unions, and a technique such as occurrence types [39] could improve precision in these circumstances.

Three additional false positives occurred because a benchmark redefined an existing method, which DRuby forbids since then it cannot tell at a call site whether the previous or the redefined method is called. (Unlike statically typed object-oriented languages, redefined methods in Ruby can have radically different types.)

The remaining false positives occurred because DRuby could not resolve the use of an intersection type; because DRuby could not locate the definition of a constant; because of a wrapper around require that dynamically changed the argument; and because of rebinding of the new method.

7. RELATED WORK

Many researchers have previously studied the problem of applying static typing to dynamic languages. Some of the earliest work in this area is *soft typing* for Scheme, which uses a set-based analysis to determine what data types may reach the destructors in a program [14, 6, 41, 18]. Typed Scheme adds type annotations and type checking to Scheme [39]. One of the key ideas in this system is *occurrence types*, which elegantly model type testing functions used to discriminate elements of union types. As mentioned earlier, DRuby support for occurrence types would be useful future work.

Several researchers investigated static typing for Smalltalk, a close relation of Ruby. Graver and Johnson [20] propose a type checking system that includes class, object, union, and block types. Strongtalk [35], a variant of Smalltalk extended with static types, has a similar system with additional support for mixins [10]. Spoon [34] and RoelTyper [42] each a present type system that trades precision for scalability. Agesen et al. [4, 3] explored type inference for Self, which is an object-based (rather than class-based) language. DRuby differs from these system as it integrates both static type inference and dynamically checked annotations.

Type inference for Ruby has been proposed before. Kristensen [25] claims to have developed a Ruby type inference system, but his thesis is not available on-line, and emails requesting it were not returned. Morrison [27] developed a type inference algorithm that has been integrated into RadRails, an IDE for Ruby on Rails. In RadRails, type inference is used to select the methods suggested during method completion. There is no formal description of RadRails's type inference algorithm, but it appears to use a simple intraprocedural dataflow analysis, without support for unions, object types, parameter polymorphic, tuples, or type annotations.

Several type systems have been proposed to improve the performance of dynamic languages. The CMUCL[26] and SBCL[31] Lisp compilers use type inference to catch some errors at compile time, but mainly rely on inference to eliminate runtime checks. Similarly, aggressive type inference [9], Starkiller [30], and a system proposed by Cannon [13] all infer types for Python code to improve performance. RPython is a statically-typed subset of Python designed to compile to JVM and CLI bytecode [7]. RPython includes type inference, though it is unclear exact what typing features are supported. One interesting feature of RPython is that it performs type inference after executing any load-time code, thus providing some support for metaprogramming.

There are several proposals for type inference for Javascript [8, 37]. The proposed ECMAScript 4 (Javascript) language includes a rich type annotation language with object types, tuple types, parametric polymorphism, and union types [22]. The challenges in typing Ruby are somewhat different than for Javascript, which builds objects by assigning pre-methods to them rather than using classes.

Aside from work on particular dynamic languages, the question of statically typing dynamic language constructs has been studied more generally. Abadi et al. [2] propose adding a type Dynamic to an otherwise statically typed language. Quasi-static typing takes this basic idea and makes type coercions implicit rather than explicit [36]. Gradual type systems improve on this idea further [33, 23], and have been proposed for object-oriented type systems [32]. Sage mixes a very rich static type system with the type Dynamic [21]. Tobin-Hochstadt and Felleisen [38] present a framework for gradually changing program components from untyped to typed.

Finally, our run-time type checks are based heavily on contracts, which were developed as the dynamic counterpart to static types to generalize and extend run-time assertions [17, 15]. An important property of contracts is to track *blame* through a program to ensure that when a contract is violated, the cause of the failure is correctly

correlated with the proper syntactic origin, which can be tricky in the higher-order case [16].

8. CONCLUSIONS

We have presented DRuby, a tool that aims to combine static and dynamic typing for Ruby. DRuby includes an expressive type language, a surface type annotation syntax that is close to the informal Ruby documentation, a powerful type inference system, and automatic cast insertion to dynamically check programmer-provided annotations. We applied DRuby to a range of benchmarks and found several latent type errors as well as a number of questionable coding practices, with a relatively low false positive rate. We believe that DRuby is a major step toward the goal of developing an integrated static and dynamic type system for Ruby in particular, and object-oriented scripting languages in general.

Acknowledgments

9. REFERENCES

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer, 1996.

[2] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically typed language. *ACM TOPLAS*, 13(2):237–268, 1991.

[3] O. Agesen and U. Hölzle. Type feedback vs. concrete type inference: A comparison of optimization techniques for object-oriented languages. In *OOPSLA*, pages 91–107, 1995.

[4] O. Agesen, J. Palsberg, and M. Schwartzbach. Type Inference of SELF. *ECOOP*, 1993.

[5] A. Aiken, M. Fähndrich, J. S. Foster, and Z. Su. A Toolkit for Constructing Type- and Constraint-Based Program Analyses. In *TIC*, pages 78–96, 1998.

[6] A. Aiken, E. L. Wimmers, and T. K. Lakshman. Soft Typing with Conditional Types. In *POPL*, pages 163–173, 1994.

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

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

[9] J. Aycock. Aggressive Type Inference. In *Proceedings of the 8th International Python Conference*, pages 11–20, 2000.

[10] L. Bak, G. Bracha, S. Grarup, R. Griesemer, D. Griswold, and U. Holzle. Mixins in Strongtalk. *Inheritance Workshop at ECOOP*, 2002.

[11] G. Bracha and W. Cook. Mixin-based inheritance. In *OOPSLA/ECOOP*, pages 303–311, 1990.

[12] K. B. Bruce, A. Schuett, and R. van Gent. Polytoil: A type-safe polymorphic object-oriented language. In W. G. Olthoff, editor, *ECOOP*, pages 27–51, 1995.

[13] B. Cannon. Localized Type Inference of Atomic Types in Python. Master's thesis, California Polytechnic State University, San Luis Obispo, 2005.

[14] R. Cartwright and M. Fagan. Soft typing. In *PLDI*, pages 278–292, 1991.

[15] R. B. Findler and M. Blume. Contracts as pairs of projections. In *FLOPS*, volume 3945, pages 226–241, Fuji Susono, JAPAN, Apr. 2006. Springer-Verlag.

[16] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *ICFP*, pages 48–59, 2002.

[17] R. B. Findler, M. Flatt, and M. Felleisen. Semantic casts: Contracts and structural subtyping in a nominal world. In *ECOOP*, pages 365–389, 2004.

[18] C. Flanagan, M. Flatt, S. Krishnamurthi, S. Weirich, and M. Felleisen. Catching Bugs in the Web of Program Invariants. In *PLDI*, pages 23–32, 1996.

[19] M. Furr and J. S. Foster. Polymorphic Type Inference for the JNI. In *ESOP*, pages 309–324, 2006.

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

[21] J. Gronski, K. Knowles, A. Tomb, S. Freund, and C. Flanagan. Sage: Hybrid Checking for Flexible Specifications. *Scheme and Functional Programming*, 2006.

[22] L. T. Hansen. Evolutionary Programming and Gradual Typing in ECMAScript 4 (Tutorial), Nov. 2007.

[23] D. Herman, A. Tomb, and C. Flanagan. Space-efficient gradual typing. *Trends in Functional Programming*, 2007.

[24] A. Igarashi and H. Nagira. Union types for object-oriented programming. In *SAC*, pages 1435 – 1441, 2006.

[25] K. Kristensen. Ecstatic – Type Inference for Ruby Using the Cartesian Product Algorithm. Master's thesis, Aalborg University, 2007.

[26] R. A. MacLachlan. The python compiler for cmu common lisp. In *ACM conference on LISP and functional programming*, pages 235–246, New York, NY, USA, 1992.

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

[28] B. Pierce. *Programming with Intersection Types and Bounded Polymorphism*. PhD thesis, CMU, 1991.

[29] B. C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[30] M. Salib. Starkiller: A Static Type Inferencer and Compiler for Python. Master's thesis, MIT, 2004.

[31] Steel Bank Common Lisp, 2008. `http://www.sbcl.org/`.

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

[33] J. G. Siek and W. Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, September 2006.

[34] S. A. Spoon. *Demand-driven type inference with subgoal pruning*. PhD thesis, Georgia Institute of Technology, Atlanta, GA, USA, 2005. Director-Olin Shivers.

[35] Strongtalk, 2008. `http://www.strongtalk.org/`.

[36] S. Thatte. Quasi-static typing. In *POPL*, pages 367–381, 1990.

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

[38] S. Tobin-Hochstadt and M. Felleisen. Interlanguage migration: from scripts to programs. In *OOPSLA*, pages 964–974, 2006.

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

[40] D. A. Wheeler. Sloccount, 2008. `http://www.dwheeler.com/sloccount/`.

[41] A. Wright and R. Cartwright. A practical soft type system for scheme. *ACM TOPLAS*, 19(1):87–152, 1997.

[42] R. Wuyts. RoelTyper, May 2007. `http://decomp.ulb.ac.be/roelwuyts/smalltalk/roeltyper/`.