The Ruby Type Checker

Brianna M. Ren John Toman T. Stephen Strickland Jeffrey S. Foster

Department of Computer Science
University of Maryland, College Park
{bren, jtoman, sstrickl, jfoster}@cs.umd.edu

ABSTRACT

We present the Ruby Type Checker (rtc), a tool that adds type checking to Ruby, an object-oriented, dynamic scripting language. Rtc is implemented as a Ruby library in which all type checking occurs at run time; thus it checks types later than a purely static system, but earlier than a traditional dynamic type system. Rtc supports type annotations on classes, methods, and objects and rtc provides a rich type language that includes union and intersection types, higher-order (block) types, and parametric polymorphism among other features. Rtc is designed so programmers can control exactly where type checking occurs: type-annotated objects serve as the "roots" of the type checking process, and unannotated objects are not type checked. We have applied rtc to several programs and found it to be easy to use and effective at checking types.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Design, Reliability, Verification

Keywords

run-time type systems, gradual typing, object-oriented type systems, Ruby

1. INTRODUCTION

Dynamic typing is a popular feature of many dynamic languages, in part because it meshes well with the goals of supporting rapid prototyping and providing a high degree of flexibility and agility to the programmer. However, dynamic typing has a major drawback: type errors can remain latent long into the software development process or even into deployment. To address this concern, there have been many proposals for adding static types to dynamic languages [9, 15, 5, 11, 3, 4, 14]. However, while these prior systems are promising, they have two key limitations. First, because they are purely static, they do not deal well with highly dynamic language features such as `eval` or reflective method invocation. Second, since static type systems must be conservative, in practice they can categorize too many programs as erroneous. Adding precision in the form of flow-, path-, and context-sensitivity helps, but also tremendously complicates the type system.

In this paper, we introduce rtc, the Ruby Type Checker, which sits at an intermediate point between pure static and pure dynamic checking. In rtc, types are checked at runtime—which is later than static typing—but at method entrance and exit—which is earlier than dynamic typing. Because rtc operates at run time, it can handle highly dynamic language features in a natural way. Moreover, as rtc only observes feasible program executions, it automatically includes the sensitivities mentioned above. Rtc is heavily inspired by and builds on the codebase of An et al's Rubydust system [1], which falls at the same design point. However, rtc is a pure type *checking* system, whereas Rubydust performs constraint-based type inference. As we discuss in various places in the paper and summarize in Section 5, this results in several key technical and implementation differences.

Rtc supports annotations on classes, methods, and objects, and rtc's type system includes nominal types, union and intersection types, block (higher-order method) types, parametric polymorphism, and type casts. A key design principle of rtc is that programmers should only "pay for what they use." That is, programs without annotations should run as usual, and programs with annotations should only perform checking where desired. To achieve this, rtc separates objects into *raw* (untyped) values and *annotated* (typed) values. Type checking only occurs when annotated values are used as receivers. Annotations are introduced either explicitly by the programmer or implicitly when values are passed as arguments to type-checked calls. We think this design strikes the right balance of providing fine enough control over type checking without requiring too much explicit annotation. (Section 2 explains the usage of rtc in more detail.)

Rtc is implemented in a similar fashion to Rubydust. Annotated objects are wrapped by proxy objects that associate types with the underlying object. When a proxy is invoked, it performs type checking before and after it delegates the method call to the underlying object. The proxy also annotates the incoming arguments and the returned value. Rtc uses some implementation tricks to maintain annotations on **self**, which would otherwise be lost when the proxy delegates to the underlying object; to handle block type checking; and to allow classes to be declared as auto-annotating, so that all instances of the class are proxied by default. (Section 3 discusses the implementation more fully.)

We evaluated rtc by adding type annotations to several small programs and running the test suites included with those programs. We found that all of the features of rtc were useful in typing our subject programs, and we were able to assign rtc types to most methods. We also found that while the overhead of rtc is substantial in relative terms, in absolute terms the test suites for our subject programs still execute quickly. (Section 4 presents our evaluation.)

In summary, we think that rtc is a practical, useful, and effective tool for increasing the type safety of Ruby programs, and that the ideas of rtc can be ported to other languages.

2. USING RTC

Figure 1 illustrates the basic use of rtc with excerpts from a payroll program with three classes: Person, the base class for describing employees of the company; Manager, a subclass of Person that includes extra information for managers; and Payroll, a class for modifying the company's payroll.

The program begins by calling require to load the rtc_lib library, which contains rtc's implementation. Next are the class definitions for Person, Manager, and Payroll. All three definitions start with a call to **rtc_annotated**, which makes annotation methods, such as **typesig**, available locally. The programmer declares types for methods by calling **typesig** with a string that contains the method name and its type. Annotating a method with **typesig** tells rtc to intercept calls to the method to perform typechecking (more on this in Section 3). For example, personnel_id (line 6) is an instance method that takes no arguments and returns the employee's id number as a Fixnum.[1] Class method from_id (line 9) takes an id number and returns the appropriate instance of Person. Finally, the manager instance method returns either the Manager of the employee or **false** if the employee has no manager; note the use of a union type on line 12 to denote these possibilities. Here, **%false** is shorthand for the class FalseClass, of which the value **false** is the only inhabitant. This and other type aliases like **%true**, **%bool**, and **%any** are used to make types both clear and concise. Rtc allows programmers to define type aliases with **typesig** "type %type_name=t", where t is some valid rtc type. After the above call %type_name may be used within the defining class wherever a type is expected.

Class Manager includes a method employees that returns an array of employees managed by the receiver. Notice that we provide the type annotation for employees on line 22 *after* its definition. We use this ability to add types to the Ruby core library without modifying its code—instead we simply reopen the core library classes as allowed by Ruby and add appropriate **typesig** calls. Although we do not show it here, **rtc_annotated** can also appear late in a class definition, but it must occur before any other rtc forms like **typesig** are used.

Often in Ruby, methods are called in several different ways. One such example is Payroll#give_raise[2] on line 30. The first two arguments to the method are the employee receiving the raise and the manager that signs off on the raise. Either id numbers or objects are allowed in both positions; however, callers may not mix the two in a given call. Thus we use an intersection type: we write multiple annotations for the same method (lines 28–29), and the resulting method type is the intersection of all such annotations. When the method is called, the arguments are checked to ensure they conform to one of the allowed patterns.

In rtc, type checking happens eagerly when a method is called, which may detect errors earlier than standard dynamic typing. For example, suppose our program passes a type-incorrect final argument to give_raise. In standard Ruby, we would need to wait until the program reaches

```
1  require 'rtc_lib'
2
3  class Person
4    rtc_annotated
5     ...
6    typesig "personnel_id : () → Fixnum"
7    def personnel_id ... end
8
9    typesig "self.from_id: (Fixnum) → Person"
10   def self.from_id(id) ... end
11
12   typesig "manager: () → Manager or %false"
13   def manager ... end
14 end
15
16 class Manager < Person
17   rtc_annotated
18   def employees
19     # ... find all managed employees in the database
20   end
21    ...
22   typesig("employees: () → Array<Person>")
23 end
24
25 class Payroll
26   rtc_annotated
27    ...
28   typesig "self.give_raise :(Fixnum,Fixnum,Fixnum)→Fixnum"
29   typesig "self.give_raise :(Person,Manager,Fixnum)→Fixnum"
30   def self.give_raise (emp, okayed_by, incr)
31     ... # ensure okayed_by is in charge of emp
32     curr = fetch_salary_from_database (emp)
33     set_salary (emp, curr + incr)
34   end
35 end
36
37 ids_1 = [1141,1231,3142] # raw, untyped value
38 ids_1.push "foo"         # allowed for raw value
39 ids_2 = [1141,1231,3142].rtc_annotate("Array<Fixnum>")
40 ids_2.push "foo"         # type error
41
42 # Assuming employee number 1141 is a Manager
43 m = Person.from_id(1141)
44 m.employees # type error
45 m_1 = m.rtc_annotate "Manager" # type error
46 m_2 = m.rtc_cast "Manager" # ok
47 m_2.employees # ok
48
49 sm = m.manager # sm: Manager or %false
50 unless sm
51   ssm_1 = sm.manager # type error
52   ssm_2 = sm.rtc_cast("Manager").employees # ok
53 end
```

Figure 1: Basic usage of rtc

[1] Fixnum is the Ruby type for fixed-size integers.

[2] Following the convention in Ruby documentation, the notation C#m refers to class C's instance method m.

line 33 to see the error—but this may take a relatively long time if the preceding database operation is slow. In contrast, rtc detects and reports the type error on entry to give_raise . In our experience with writing Ruby programs, we are often frustrated with exactly this problem: while programs can be quick to write, they often contain small, frustrating mistakes that manifest late.

One design goal of rtc is allowing programmers to use types where desired and eschewing type checking elsewhere. Thus, rtc employs a finer grained strategy than Rubydust [1], in which developers decide on a per-class basis whether to use types. In rtc, newly created objects, dubbed *raw* objects, are untyped by default, so invoking their instance methods does not involve type checking. For example, even though rtc contains type annotations for the Array class, a newly created array is initially untyped (lines 37–38). There are two ways to enable type checking for a given value. First, the programmer can use rtc_annotate to create an *annotated* version of a value that carries a type (line 39). When an annotated value is the receiver of a call to a type-annotated method, rtc performs type checking (line 40). Second, when any value, raw or annotated, is passed as an argument or returned from a method for which rtc performs type checking, then rtc checks that the value is consistent with the declared type. If the value is already typed, then rtc checks that the current type is a subtype of the desired type, raising a type error if it is not, and then rewraps the contained value with the desired type. If the value is not typed, then rtc checks first-order properties of the value, such as its class, to determine whether the value is consistent with the desired type. If it is not, then rtc raises a type error. If it is consistent, then within the method body rtc annotates the value with the declared type. For example, when our program calls Person.from_id[3] on line 43, the value 1141 is annotated with the type Fixnum within the body of Person.from_id.

Unlike instance methods, rtc checks every call to annotated class methods, such as the call to Person.from_id on line 43. We choose to have rtc automatically check class methods to reduce the annotation burden; forcing the programmer to write C.rtc_annotate (...). m to get type-checked class methods would be a large change to existing programs.

In addition, rtc assumes a subclass is a subtype of its superclass by default; the programmer can annotate a class with no_subtype if this is not the case. Thus, it is possible for objects to become annotated with proper supertypes of their actual, run-time type. For example, on line 43 we use Person.from_id to get employee 1141 on line 43. While we may know this employee is a manager, Person.from_id is annotated to return a Person. Thus, our program cannot call the employees method directly on the result (line 44).

One design choice would be to allow rtc_annotate to perform a downcast. However, since this operation is conceptually different than an upcast, we prefer to use a distinct method call. Thus, we restrict rtc_annotate (line 45), and similarly the re-annotation that occurs at method entry, to only safe upcasts; and rtc provides rtc_cast for cases where the programmer desires a downcast during reannotation (lines 46–47). The method rtc_cast is particularly useful when working with union types. For example, on line 49 the variable sm may contain either a Manager or **false**. Our program uses **unless** to test for falsity, so on lines 51–52 we know that sm is a Manager. However, since rtc's implementation cannot automatically reassign types based on conditions (see Section 3), we must add an explicit use of rtc_cast to reflect this knowledge in the program.

[3]Class method m of class C is referred to as C.m.

Next, we discuss some of the key features of rtc, particularly places where type checking differs from inference significantly, or features that are lacking in Rubydust.

Blocks and procedures. Ruby supports higher-order programming through the use of *code blocks*, which are anonymous methods passed in using a special syntax. Code blocks are not first-order objects (they can only be called using the special **yield** expression), but blocks can be freely converted to Proc objects, which are first class.

As an example, the String class defines method each_char, which calls its block argument on each character of the receiver as a string of length 1. Rtc includes the following type annotation for each_char:

```
class String
  rtc_annotated
  typesig "each_char: () { (String) → %any } → String"
end
```

Here, since the return value of the block is not used by each_char, we use the type **%any** to signify that the block may return any value.

Blocks types were not supported in Rubydust. Rtc implements block typing by wrapping block arguments in a Proc object that does type checking on entry to and exit from the block; more details are in Section 3.

Parametric Polymorphism. Rtc supports parametric polymorphism for classes and methods. For example, here are some of the annotations already included in rtc on the built-in Array class:

```
58 class Array
59   rtc_annotated [:t, :each]
60
61   typesig " '[]': (Range) → Array<t>"
62   typesig " '[]': (Fixnum, Fixnum) →Array<t>"
63   typesig " '[]': (Fixnum) → t"
64
65   typesig "map<u>: () {(t) → u } → Array<u>"
66 end
```

On line 59, we call **rtc_annotated** and include a two-element list argument to indicate the Array class should be parameterized by its contents type. The first element of the list, :t, names the type parameter. The second element, :each, indicates how to find the contents type of :t for a raw Array. More specifically, when checking whether a raw array can be annotated with a type Array<u>, rtc will call the each method to iterate over all the array elements and check whether they are compatible with type u. Classes with multiple type parameters can be specified by passing multiple two-element list arguments to **rtc_annotated**.

Note that iterating through raw arrays is potentially very expensive, and so rtc includes a *non-strict mode* that omits it (see Section 3 for details). Additionally, in some cases, classes may have type parameters that are cannot be inferred by iterating over the contents. For these cases, the programmer can omit the iterator method name when calling **rtc_annotated**; rtc signals a type error if a raw instance of such a class is passed to a typed position.

Lines 61–63 give the type for one commonly used method, the array getter []. Note the type parameter t is in scope inside the class, so it can be used in these annotations.

Line 65 illustrates method polymorphism with the type for map. For this method, rtc attempts to infer the instantiation of u at a method call. For many polymorphic methods we can infer the right instantiation by examining the arguments when the method is invoked. For map, however, it is slightly trickier, as rtc cannot know the return type of the block until it is called. In this case, rtc assigns u to the type of the value returned by the first call to the block, or to %none (the bottom type) if the block is never called. Further returns from the block are checked using this inferred type, and when map returns, rtc checks that the returned array is type Array<u>.

This approach to type inference may not choose the correct types for instantiation, however. Consider the following use of Array#*map*, where the block returns numbers for even inputs and strings for odd inputs:

```
a = [1,2,3]. rtc_annotate("Array<Fixnum>")
a.map() { |n| if (n % 2 == 0) then n else n.to_s end }
```

In the above example the call will fail because our type checker infers u to be the type String from the first use of the block, but the block returns a Fixnum from its second use. To address this issue, rtc includes a method rtc_instantiate to explicitly instantiate type parameters. In this case, the instantiation returns a method object of the correct type:

```
m = a. rtc_instantiate (:map,:u⇒"Fixnum or String")
m.call() { |n| if (n % 2 == 0) then n else n.to_s end }
```

Ambiguity in union and intersection types. While union and intersection types are heavily used in type annotations, rtc must forbid some uses that are problematic from a type checking perspective. For example, recall the Person and Manager classes from figure 1 and consider the following intersection type:

```
typesig "seat: (Person) → Cubicle"
typesig "seat: (Manager) → Office"
```

If we pass in a Manager, both arms of the intersection are valid since Manager is a subtype of Person. We could choose various disambiguation rules, but to keep rtc simple and predictable we opt to report an error when such an ambiguously typed method is called.

Type variables can also introduce ambiguity. For example:

```
typesig "m1<t,u>: (t or u) → Array<t> or Hash<String, u>"

typesig "m2<t>: (t) →Array<t>"
typesig "m2<u>: (u) →Hash<String, u>"
```

The uses of t and u above are ambiguous because they appear in the same place in a union or intersection type. Thus, rtc forbids such type annotations by reporting an error when an ambiguously typed method is called.

Similarly, having a concrete type and a type variable at the same level causes ambiguity:

```
typesig "m3<t>: (t or Fixnum) →Array<t>"
```

If a value of type Fixnum is provided, then we cannot determine whether type variable t should be assigned Fixnum or whether we are using the other branch of the union and t should be some other type. (Here we see that **or** is regular union, rather than disjoint union.)

Note that not all uses of type variables create ambiguity:

```
typesig "m4<t,u>: (Array<t> or Hash<String, u>) →t or u"

typesig "m5<t>: (Array<t>) →t"
typesig "m5<u>: (Hash<String, u>) →u"
```

In these annotations, we can determine the bindings of the type variables depending on whether the argument is a Array or Hash. In the case of m4, the type variable in the unused part of the union gets assigned the empty type %none.

Tuple types. Ruby programmers often use Arrays both homogeneously for unbounded lists, and heterogeneously for fixed-size tuples. Like DRuby, rtc includes a special type Tuple<t1,..., tn> representing an array whose ith element has type ti [9]. Values of type Tuple can be manipulated using a subset of the Array methods that do not change the size of the array or the order of array elements. For example, Array#*[]* (element access) is allowed, but Array#*push* is not. Note that this is different than DRuby, which performs inference and thus begins by assuming every array literal is a Tuple and then promotes it to an Array if non-Tuple methods are used on it.

Instantiating proxy objects automatically. Sometimes a programmer may want all instances of a given class to be annotated without having to explicitly call rtc_annotate. To achieve this, the programmer adds a call to rtc_autowrap in an annotated class definition. Classes that are subtypes of an auto-wrapping class are also auto-wrapping. Currently, auto-wrapping works only with non-parameterized classes.

3. IMPLEMENTATION

We have implemented rtc as a Ruby library. Rtc adds rtc_annotate, rtc_cast, and other key methods to the base Object and Class classes as appropriate, so they are available everywhere. When the programmer uses rtc_annotate to add a type to an existing object, rtc wraps the original object in a *proxy* that also contains the type. The proxy defines a method_missing method, which in Ruby receives a call when calling an undefined method on the object.[4] Calls to the proxy first ensure the arguments are of the appropriate type, then delegate to the original object, and finally check the return value's type before returning to the callee. The general idea of proxy wrapping is borrowed from Rubydust, though rtc does not perform constraint generation [1].

In more depth, consider the code at the top of Figure 2; the bottom part of the figure shows the objects resulting from this code. On line 83, we annotate the array from line 82 with the type Array<Object>. This annotation returns a new instance of the internal rtc class Proxy that holds both the underlying object and its type. Similarly, line 85 assigns a new Proxy to n.

[4]Since calling methods via method_missing is slower than direct dispatch, we explicitly delegate some Object operations such as ==, **class**, or **nil**? due to their prevalence.

```
82  a = [1,2,3]
83  b = a.rtc_annotate("Array<Object>")
84  # equiv. to b = Proxy.new(a, "Array<Object>")
85  n = 4.rtc_annotate("Fixnum")
86  b.push(n)
87  # the array now contains 1, 2, 3, and Proxy(4, "Object")
88  m = b[1]
89  # m is bound to the value Proxy(2, "Object")
```

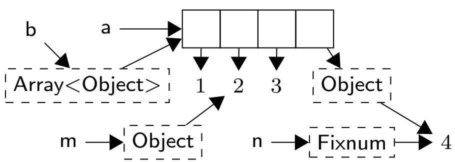
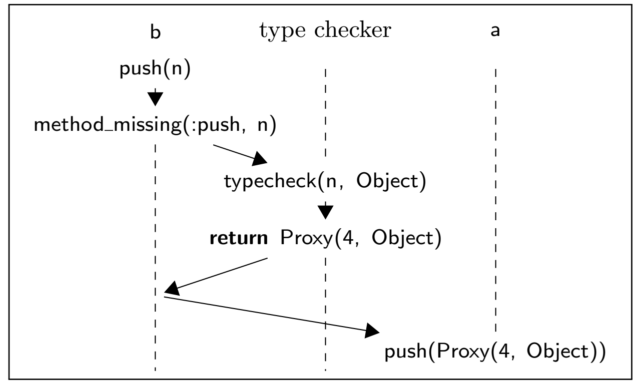
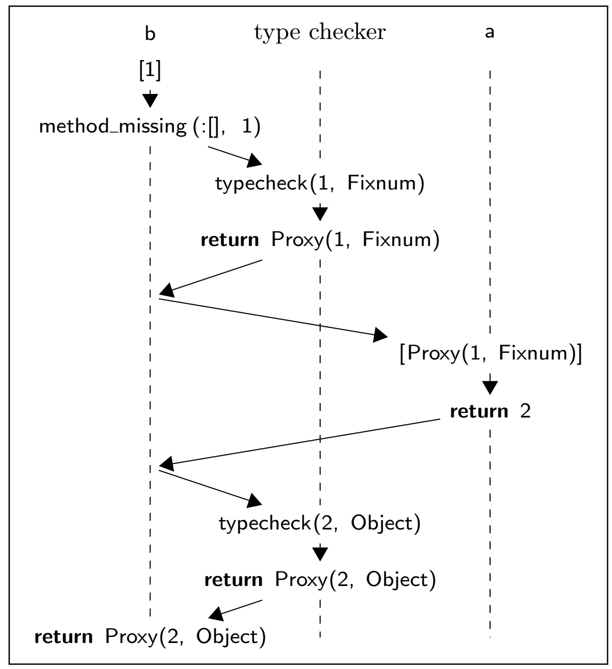


Figure 2: Illustration of proxy implementation.



(a) Sequence diagram for line 86



(b) Sequence diagram for line 88

Figure 3: Illustration of proxy implementation (cont'd).

Next consider the call on line 86; the sequence of events triggered by this call is shown in Figure 3a. When push is invoked on the proxy object b, Proxy#method_missing is called with two arguments: :push, the name of the method, and n. Then method_missing checks the type of the argument by retrieving the type of the push method and comparing the type of the argument against the expected type. Rtc then rewraps the underlying object in a new Proxy with the formal argument type and returns the new proxy to method_missing. This ensures that the method must use the value according to the method's type signature (here, Object) instead of its possibly more specific type (here, Fixnum). Finally, the rewrapped argument is passed to the underlying array's push method, which adds it to the end of the array a.

Next consider line 88, which sets m to b[1]; Figure 3b illustrates this call. As before, b's method_missing receives the call to []. This time, the argument 1 is a raw value, so rtc retrieves the value's class to derive its type. Since there is only one argument of class Fixnum, rtc infers that this call uses [] with the type (Fixnum)→Object. The type checking algorithm then wraps 1 in a Proxy with the type Fixnum before it is passed to the underlying object's [] method. Similarly, the unannotated value 2 in the array is wrapped in a Proxy with type Object (the inferred return type of []) before it is returned from the call.

As we have just seen, adding annotations to objects means that annotations get added to method arguments and results, even if those values were originally unannotated. Operations on those newly annotated values can add further annotations. Thus, the user need not annotate all objects explicitly to get wide type checking coverage, but rather can annotate just a few key objects to get the ball rolling.

Type checking blocks and procedures. Thanks to Ruby's support for higher-order procedures, type checking blocks and procedures is straightforward. If the value to type check is a block, rtc first converts it to a procedure, and otherwise we use the procedure value directly. Next, rtc creates a new procedure that first checks the arguments, then calls the original procedure, and finally checks the return value from that call. This conversion is similar to the conversion performed by Findler and Felleisen [7] to protect higher-order functions with contracts. If the original value is a procedure, then rtc uses the new procedure as the new value, and otherwise, rtc then converts the resulting procedure back into a block before use.

Type checking in method wrappers. As we will explain shortly, we need to add another layer of interposition to track proxies on **self** and to support calls to native methods. Thus, rtc alters annotated classes to add a *method wrapper* layer within annotated classes themselves. To implement this alteration, rtc uses some low-level features of Ruby to rename methods in the original object to a mangled name. It then inserts a new method with the original name that delegates to the original method. Rtc adds a method instead of inserting a generic method_missing for improved performance. It is in these method wrappers that rtc performs type checking if a Proxy received the previous call.

In developing this implementation, we discovered one interesting quirk of Ruby. There are two ways to define new methods: using define_method, which takes a Proc object as an argument, or using eval. We found that methods created by the former mechanism are much slower to call than methods created by the latter. Thus, we use eval to create new methods although it is less elegant.

Due to this design, calling an annotated method in rtc entails two method interceptions: one in the Proxy and one in the method wrapper layer. To improve performance in the method wrapper layer, we directly call the (name-mangled) original methods of underlying objects that rtc uses internally in its type checking process.

Tracking proxies on self. When a Proxy finally delegates to the underlying object's method, **self** will be bound to the underlying object rather than the Proxy. Thus, if that method in turn invokes other methods on **self**, without further work we will fail to type check those calls, since the receiver will not be a Proxy.

We solve this problem using the method wrapper layer. Internally, rtc maintains a stack of Proxys associated with each object. The method_missing of a Proxy pushes **self** (that is, the proxy) onto the stack associated with the wrapped object before delegating and pops the stack after normal or exceptional exit of the delegated method. When an annotated method is intercepted by the method wrapper layer, it also checks whether there is a Proxy on the stack. If so, it performs type checking using the type information contained in the topmost Proxy. This ensures type checking continues to occur for calls targeting **self** in annotated objects.

Handling methods that expect native values. Certain methods of built-in types—particularly those implemented in native code—expect their arguments to be objects of an appropriate class, and passing in Proxys instead causes those operations to fail. Thus, **typesig** optionally takes an :unwrap argument that is an array of argument positions from which the proxy must be removed before calling the method. For example, we can annotate the + operation on Fixnum to unwrap its argument:

```
typesig "'+':(Fixnum)→Fixnum", :unwrap⇒[0]
```

In the method wrapper layer, we remove proxies as specified by :unwrap before calling the original method.

Handling **false** *and* **nil**. A related problem is that boolean comparisons and conditional expressions, which cannot be intercepted in Ruby, treat **false** and **nil** as false and all other values as true. Thus, wrapping **false** or **nil** in a proxy would cause them to be treated as true, yielding incorrect results. As a result, we do not wrap either these values in proxies during type checking.

Non-strict mode. As discussed in Section 2, rtc checks that raw objects have the correct type whenever they are annotated; for container classes like Array, this check involves iterating over the contents, which can be quite expensive.

Thus, rtc includes a *non-strict mode* in which this iteration is omitted. That is, in non-strict mode, when raw values are annotated only the type constructor is checked for compatibility, but not the type parameters. For example:

```
# non−strict mode
[1,2,3]. rtc_annotate("Fixnum")            # error
[1,2,3]. rtc_annotate ("Array<String>")  # ok
```

While non-strict mode does not catch errors as soon as possible, errors are caught on uses of the contained values. For example, consider the following code:

```
# non−strict mode
class Statistics
  rtc_annotated
  typesig "sum: (Array<Fixnum>) →Fixnum"
  def sum(input)
    total = 0
    input.each { |elem|
      total += elem
    }
    total
  end
end
Statistics .new.rtc_annotate(" Statistics ").sum(["1", "2", "3"])
```

In non-strict mode, the argument to sum is accepted although the contents do not match the expected type. However, rtc deduces from its annotation that the block argument to each accepts type Fixnum. When the first element of the array, of type String, is passed to the block, the block wrapper checks it against type Fixnum and reports an error.

In Section 4, we compare the performance of strict and non-strict modes. As the latter is significantly faster that than former, non-strict mode is enabled by default. However, the programmer may opt-in to stricter type checking by setting the global variable $RTC_STRICT to **true**.

4. EVALUATION

We performed an initial evaluation of rtc on a set of Ruby programs and libraries that we retrofitted with rtc types. Figure 4 summarizes the results. The subject programs are as follows:

- Sudoku: an implementation of Norvig's algorithm for solving Sudoku puzzles.
- Ascii85: a program for encoding/decoding Adobe's binary-to-text encodings of the same name.
- ministat: a library that computes statistical info such as mode, median, mean, variance, etc.
- finitefield: an implementation of finite field arithmetic.
- hebruby: a Hebrew data conversion program.
- set: Ruby's set library and its associated test cases.
- Ruby Data Structures (abbreviated RDS): a library of common data structures. We annotated two classes, SinglyLinkedList and SinglyLinkedListElement.

The first five of these programs come from the Rubydust benchmark suite [1]. We also tried to annotate the other three Rubydust benchmarks, but those programs fail to run under the latest version of Ruby, which rtc requires.

In addition to annotating the subject programs, we also annotated the built-in Array, Hash, and Set libraries. These particular libraries were chosen because they are the basis for most user-defined data structures and they saw the heaviest use in the programs we used for our evaluation.

Next we report on the overhead of rtc, which rtc features were used for the subject programs, and the ease of the conversion process.

Efficiency. The first three columns of the Figure 4 report the running times for the program's test suite on the original program; on the annotated program under non-strict mode; and on the annotated program under strict mode. While

program	time (s) unann.	non-strict	strict	annotated methods	unann. methods	annotated values	Features: Tuple	$\{\cdot\}$	(τ)	$\cup$	$\cap$	$\forall$
Sudoku-1.4	0.04	5.34	7.58	8	1	10	0	0	2	5	0	0
Ascii85-1.0.2	0.02	0.05	0.05	2	1	0	0	0	0	0	0	0
ministat-1.0.0	<0.01	0.30	0.56	13	1	0	0	0	0	0	0	0
finitefield-0.1.0	<0.01	0.02	0.02	10	1	0	0	0	0	0	0	0
hebruby-2.0.2	<0.01	0.12	0.12	19	1	0	1	0	0	1	0	0
RDS-1.0.0	<0.01	0.01	0.01	7	2	3	0	0	0	3	0	7
library												
Array	–	–	–	71	4	–	0	28	–	7	35	18
Hash	–	–	–	38	2	–	4	12	–	5	9	8
Set	–	–	–	21	13	–	0	7	–	0	2	7

Features: Tuple = Tuple types, $\{\cdot\}$ = block types, (τ) = rtc_cast, $\cup$ = union types, $\cap$ = intersection types, $\forall$ = polymorphic types

Figure 4: Summary of evaluation results

the performance overhead of rtc is relatively large, the test suites run quite rapidly in most cases, suggesting that rtc is practical in many testing scenarios. As mentioned in section 3, rtc creates a wrapper layer that all calls must go through whether there is an active proxy or not. This extra level of indirection is the main source of the overhead in rtc due to the inefficiency of method calls in Ruby.

The program with the most substantial overhead is Sudoku. This program makes extensive use of large arrays and hashes, and so the overhead of rtc's method interception has a large cost. To partially address this issue, rtc can be disabled in production by setting by the RTC_DISABLE environment variable to a non-empty value. When rtc is disabled, no wrappers are created by calls to **typesig**. In addition, annotations on objects via rtc_annotate and rtc_cast become no-ops. That is, instead of returning a new proxy object, they simply return **self**. This enables the programmer to use rtc in a test environment where some overhead may be acceptable and then disable rtc in the field.

Rtc features. The middle three columns of Figure 4 count the number of unannotated methods, annotated methods, and explicit annotations of values we added. In the subject programs, the only unannotated method was initialize, the constructor. Since the receiver of initialize is always a newly created, and hence raw, object, rtc will never type check an initialize call. In the future, we plan to investigate other policies for constructors.

The only subject program for which we needed explicit rtc_annotate calls was Sudoku. These annotations were for several large arrays built during initialization, and they improved the performance of rtc in strict mode since otherwise rtc would repeatedly iterate over those arrays to infer their types at each use.

In the library classes, we were able to annotate almost all of the methods for Array and Hash. There were several methods, however, that have no reasonable type annotation in rtc. For example, Array#flatten returns a new array in which arbitrary depth nestings of array have been removed from the receiver. Even worse, Array#flatten! does the same, but mutates the receiver object. We leave these as unannotated, so they may be used but are not type checked. Unannotated methods in the Set class include Set#flatten and methods that use the Enumerable class, which is a mixin for collection classes; rtc currently does not support mixins.

The rightmost columns of Figure 4 list how often various typing features of rtc were used. To count uses of polymorphic types, we counted how many classes or methods have annotations that bind type variables. The most commonly used features in the subject programs are union and polymorphic types, and the most commonly used features in the Ruby standard libraries are intersection and block types. There were very few uses overall of tuple types and rtc_cast.

The annotation process. We found the process of annotating the subject programs to be relatively straightforward: we examined their code, looked for program invariants assumed by the original authors, and turned those invariants into type annotations. Although this was somewhat time consuming for us, we expect the original authors of the methods would be faster at this process.

We often had to iterate the annotation process as we found mistakes in our **typesig**s. The most common errors we made fall into a few groups. For some programs, we missed edge cases in methods, such as sometimes returning **false**, and so would initially annotate a method with a subset of its possible return values. Similarly, we sometimes missed an arm of an intersection type. Finally, we sometimes forgot to cast a value typed with a union type to a more specific type after the value was tested. In all cases, we found our errors immediately upon running the test suites under rtc.

Not all type errors were due to our mistakes, however. The Sudoku solver contains the following (correct) annotations:

```
typesig "search: (...) → %false or Hash<String,String>)")
typesig " string_solution : (Hash<String,String>) → String")
```

The search method returns **false** if the given puzzle is impossible to solve, while string_solution assumes that it is given a valid puzzle solution. In the test suite, the return of search is fed directly into string_solution without checking for **false**. While an error due to this mismatch never happens in the test suite because all its puzzles are solvable, rtc appropriately raises a type error.

5. RELATED WORK

As discussed in the introduction, rtc builds on the Rubydust system of An et al. [1]; we even reuse some of the same code base, specifically the type language parser and some of the proxy-related code. The key difference is that rtc is a type checking system, whereas Rubydust performs type inference. The addition of checking introduces several new concerns: adding explicit annotations (rtc_annotate) and type casts (rtc_cast); inferring types of raw values passed to annotated positions; and making control of type checking finer grained, that is, driven by annotated objects, rather than by annotations on classes as in Rubydust. Rtc also supports some features that Rubydust does not, including block types and tracking type annotations on **self**. Rubydust does not do the latter because its coarse-grained distinction between typed and untyped code means it does not matter whether **self** is proxied. Finally, perhaps the most important difference from a usability perspective is that rtc type errors are reported as soon as they occur, whereas Rubydust generates constraints and only solves them at the end of execution. Thus, it may be harder in Rubydust to understand reported errors.

Several researchers have proposed adding static types and static type inference to various dynamically typed languages, including Ruby [9, 8], Python [5, 11, 3], and JavaScript [4, 14] among others. Similarly, gradual type systems [12] like those for Scheme [15] and Thorn [6] pair a dynamically typed language with a sister, statically typed language. The typed and untyped parts of a program are allowed to interact without breaking the invariants of the typed language. All of these systems perform static analysis, whereas rtc is a library that operates purely at run time. One advantage of rtc's approach is that it does not require maintaining a Ruby frontend, which the Rubydust authors have pointed out as problematic [2]. Another advantage of rtc is that because it operates at run time, rtc only observes realizable execution paths through the target program and can easily operate in the presence of dynamic features such as eval, reflective method invocation, and method_missing.

Rtc's dynamic implementation is inspired by research into contract systems. Existing contract systems for Ruby are limited to "design by contract" [10] systems, which annotate classes with preconditions, postconditions, and invariants that are simple assertions checked only on method entry and method exit. Rtc's dynamic checks are closer to those provided by higher-order contracts [7, 13]. Like higher-order contract systems, rtc wraps method arguments and results with proxies that stay with those objects as they flow through the program. This enables rtc not just to enforce preconditions and postconditions, but also to check that the type of a parameter is adhered to within the body of a method and that the type of a return value is respected long after the method has returned.

6. CONCLUSION

We present rtc, a Ruby library that adds type checking at method call boundaries. Rtc uses proxy objects to wrap regular objects with annotated types and only type checks annotated methods on classes and proxied objects. Our experimental results suggest that rtc is a practical, useful system. In the future, we plan to apply rtc to Ruby on Rails programs, and explore extending its type checking capability to reason about some of the complex invariants of the Rails framework.

Acknowledgments

Thanks to Aseem Rastogi, Mike Hicks, and the anonymous reviewers for their comments on earlier drafts of this paper. This research was supported in part by NSF CCF-0915978 and CCF-1116740.

7. REFERENCES

[1] Jong-hoon (David) An, Avik Chaudhuri, Jeffrey S. Foster, and Michael Hicks. Dynamic Inference of Static Types for Ruby. In *POPL*, 2011.

[2] Jong-hoon (David) An, Avik Chaudhuri, Jeffrey S. Foster, and Michael Hicks. Position Paper: Dynamically Inferred Types for Dynamic Languages. In *STOP*, 2011.

[3] Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas Matsakis. RPython: Reconciling Dynamically and Statically Typed OO Languages. In *DLS*, 2007.

[4] Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards Type Inference for JavaScript. In *ECOOP*, 2005.

[5] John Aycock. Aggressive Type Inference. In *International Python Conference*, 2000.

[6] Bard Bloom, John Field, Nathaniel Nystrom, Johan Östlund, Gregor Richards, Rok Strniša, Jan Vitek, and Tobias Wrigstad. Thorn: Robust, Concurrent, Extensible Scripting on the JVM. In *OOPSLA*, 2009.

[7] Robert Bruce Findler and Matthias Felleisen. Contracts for Higher-Order Functions. In *ICFP*, 2002.

[8] Michael Furr, Jong-hoon (David) An, and Jeffrey S. Foster. Profile-Guided Static Typing for Dynamic Scripting Languages. In *OOPSLA*, 2009.

[9] Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, and Michael Hicks. Static Type Inference for Ruby. In *OOPS Track at SAC*, 2009.

[10] Bertrand Meyer. Applying Design by Contract. *IEEE Computer*, 25(10):40–51, October 1992.

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

[12] Jeremy G. Siek and Walid Taha. Gradual Typing for Functional Languages. In *Scheme and Functional Programming Workshop*, 2006.

[13] T. Stephen Strickland and Matthias Felleisen. Contracts for First-Class Classes. In *DLS*, 2010.

[14] Peter Thiemann. Towards a Type System for Analyzing JavaScript Programs. In *ESOP*, 2005.

[15] Sam Tobin-Hochstadt and Matthias Felleisen. The Design and Implementation of Typed Scheme. In *POPL*, 2008.