Refinement Types for Ruby

Milod Kazerounian, Niki Vazou, Austin Bourgerie, Jeff Foster, Emina Torlak[1]

University of Maryland [1]University of Washington

Abstract. We adjust refinement types to reason about Ruby programs that use *metaprogramming* and *dynamic mixins*. To do so, we develop refinement types for Ruby (RTR), by refining with logical specifications the types of RDL [22], a Ruby just-in-time type checker. To check the specifications, we translate Ruby programs to the verification language Rosette. In this paper, we formalize RTR and use it to check real Ruby and Ruby on Rails applications.

Keywords: ruby, rossette, refinement types, dynamic languages

1 Introduction

Refinement types combine types with logical predicates to encode program invariants [24, 35]. For example, the refinement type

$$\textsf{Integer}\ \mathsf{x}\ \{\ 0 < \mathsf{x} \leq 60\ \}$$

represents **Integer** expressions x that satisfy the predicate $0 \leq \mathsf{x} \wedge \mathsf{x} < 60$. This type can be used to precisely describe the integer expressions that represent a second (*i.e.,* unit of time). Refinement function types are used to encode pre- and postconditions (*i.e.,* contracts). For example, a function incr_sec that increases a second, is given the below refinement type specification

$$\textsf{type}\ \mathsf{incr\_sec}\ \ \text{`}(\textsf{Integer}\ \mathsf{x}\ \{\ 0 \leq \mathsf{x} < 60\ \}) \rightarrow \textsf{Integer}\ \mathsf{r}\ \{\ 0 \leq \mathsf{r} < 60\}\text{'}$$

With this specification incr_sec can only be called with integers that are valid seconds and will always return valid seconds. Refinement types were introduced to reason about simple invariants, like safe array indexing [35], but since then they have been successfully used to verify sophisticated properties including termination [31], program equivalence [1], and correctness of cryptographic protocols [20], in various languages (*e.g.,* ML [10], Racket [13], and TypeScript [32]).

But, applying refinement types to Ruby is challenging because Ruby is designed to be a very flexible, expressive, and dynamic programming language. Two key challenges for verification in Ruby are dynamic *mixins*, *i.e.,* method environments are defined at runtime, and *metaprogramming*, *i.e.,* execution relies on code generated at runtime, since they both allow method definitions to depend on methods that are not determined, or even defined, statically. Thus, verification should proceed under partial runtime environments. These challenges are magnified in Ruby on Rails (just "Rails" from here on), a popular web development framework which makes heavy use of metaprogramming.

In this paper, we introduce refinement types for Ruby (RTR), a new system that uses two key ideas to address the above challenges (§ 3). RTR uses:

– *Assume-guarantee checking* to reason about mixins. Assume-guarantee reasoning [12] allows modular verification of each method definition, independently. Each method only assumes it's preconditions and guarantee it's postcondition, irrespective of the environment under which it is executed. Under this reasoning, we verify definitions of mixin Ruby modules assuming refinement type specifications for all undefined, external methods. The exact environment under which the mixin module gets binded is irrelevant for verification, as long as each external definition in the runtime environment assumes it's preconditions and guarantees it's postconditions (§ 3.1).
– *Just-in-time checking* to reason about metaprogramming. Just-in-time type checking [22] brings static type checking into dynamic environments by statically checking types at run time. The typing environment is maintained and updated during execution and before a method is called, its body is statically type checked in the current dynamic type environment. In RTR we extend this idea to just-in-time program verification and check a method's specification at runtime, just before the method is called. With this technique, program invariants are checked early enough (*i.e.,* before they are actually violated) while our verifier allows for mataprogramming (§ 3.2).

To verify refinement type specifications, RTR translates specifications and programs at runtime into Rosette [30], a language with verification constructs, such that the Ruby program matches its specification only when Rosette is able to verify the assertions of the translated program. The translation encodes objects as structures and method calls as function calls. Following Dafny [14], we increase the precision of the translation under object mutation, by labeling methods as pure (*i.e.,* no mutation occurs) or impure (*i.e.,* objects may get mutated) and declaring the fields that cannot get modified by impure methods (§ 2).

To formalize RTR we define refined Ruby as an object-oriented core calculus λ^{RB} that allows for refinement type specifications with mutation labels. We define the translation of λ^{RB} to Rosette and claim that if all labels are sound and all refinements are pure, RTR is also sound (§ 4).

Finally, we implemented our technique by adding logical refinements to the types of RDL [22], a just-in-type type checker for Ruby programs. We used the implementation to check functional correctness, array and numeric invariants on six real Ruby and Rails applications (§ 5). In RTR verification is called per method (instead of per module), thus verified and unverified methods can coexist and depend on each other, allowing for gradual verification. This, combined with the reasonable amount of the required annotations (we used 64 specifications to verify 214 lines of Ruby), makes RTR a usable Ruby verifier.

2 Overview

We start with an overview of RTR, an extension to the Ruby type checker RDL with refinement types. In RTR program invariants are specified with refinement types (§ 2.1) and checked by translation to Rosette (§ 2.2). We translate Ruby objects and Rosette stucts (§ 2.3) and method calls to function calls (§ 2.4).

2.1 Refinement Type Specifications

Refinement types in RTR are Ruby types extended with logical predicates. For example, the type **Integer** x { $0 \leq x \wedge x < 60$ } we saw in § 1 refines **Integer** types with the Ruby expressions $0 \leq x \wedge x < 60$. Refinements can be arbitrary Ruby expressions that are *booleans* and are *pure*, *i.e.,* have no side effects, since effectful predicates make verification either complicated or imprecise [33].

We use refinement type specifications, *i.e.,* refined function types, to extend the expressiveness of types in RDL [22], a just-in-time Ruby type checker. Specifically, we use RDL's **type** method to link Ruby methods with their specifications. For example, below we define the method incr_sec (x) that takes as input an integer representing a second and returns the succeeding second, *i.e.,* resets if x == 60.

```
type '(Integer x { 0 ≤ x < 60 }) → Integer r { 0 ≤ r < 60}'
def incr_sec (x)
  if (x==59) then 0 else x+1 end
end
```

Before the definition of the method incr_sec , we call the **type** method with incr_sec 's specification, represented as a Ruby string, illustrating that the result of the method will be in the range from 0 to 60 if its argument falls in the same range. This string is parsed into RDL's type system, and stored in a global table which maintains the program's type environment.

2.2 Verification using Rosette

To check if methods satisfy their specifications RTR relies on Rosette [30], a verification oriented extension to Racket that uses symbolic execution to generate logical constraints, and discharges these constraints using Z3 [16]. Concretely, RTR generates Rosette verification queries that are valid if and only if the Ruby specifications are satisfied.

For example, to check the safety of incr_sec 's specification, RTR defines the Rosette function incr_sec using its Ruby definition and then verifies that, assuming incr_sec 's precondition, its postcondition always holds.

```
(define (incr_sec x) (if (x==59) then 0 else x + 1))
(verify assume(0 ≤x < 60);
        assert(let r = incr_sec x in 0 ≤ r < 60)))
```

We use Rosette's **verify** function with assumptions and assertions to encode pre- and postconditions, respectively, while we bind the method's result to a variable r to avoid multiple computations.

2.3 Encoding & Reasoning about Objects

We encode Ruby objects in Rosette with a struct type we name **object**, as standard [11, 26]. The struct **object** contains an integer classid identifying the

object's class, an integer objectid identifying the object itself, and a field for each instance variable of all objects encountered in the source Ruby program.

For example, consider a Ruby class **Time** with three instance variables @sec, @min, and @hour, representing the time components, and a method is_valid that checks (in-bounds) validity for all the three time components.

```
class Time
  attr_accessor :sec, :min, :hour

  def initialize (s, m, h)
    @sec = s; @min = m; @hour = h
  end

  type '() → bool'
  def is_valid
    0 ≤ @sec < 60 ∧0 ≤ @min < 60 ∧0 ≤ @hour < 24
  end
end
```

When translating the method is_valid into Rosette, the **object** struct will contain five fields that represent the class ID, object ID and the three time components.

```
(struct object ([ classid ][ objectid ]
                [@sec #:mutable] [@min #:mutable] [@hour #:mutable]))
```

The call to **struct** defines a new **object** type, along with a constructor, and getters and setters for fields of instances of that type. So, the Ruby call **Time**.new(4,6,8) will be represented as (**object** 1 2 4 6 8), assuming the class ID for **Time** is 1 and the object's ID is 2.

We use the above encoding to represent in Rosette all Ruby objects, apart from the primitive ones. We encode primitive objects, like booleans, numeric types, and arrays to their respective primitive Rosette datatypes, to achieve precise reasoning over these theories (§ 4.4). Note that since the fields of the **object** stuct are statically defined, our encoding cannot handle dynamically generated instance variables, which we leave as future work.

Verification on objects Using the **object** struct, reading from or writing to an instance variable in Ruby is translated as respectively getting or setting the field of an **object** in Rosette. As an example, we add a method mix to the **Time** class that returns a **Time** object by mixing its three **Time** arguments.

```
type :mix, '(Time t1 { t1. is_valid },
             Time t2 { t2. is_valid },
             Time t3 { t3. is_valid })
           → Time r { r. is_valid }'
def mix(t1,t2,t3)
  @sec = t1.sec; @min = t2.min; @hour = t3.hour; self
end
```

We specify that mix is only called with valid time arguments and generates a valid result. Verification fails at the basic type checking stage, since the types of field getters and setters (*e.g.,* sec and sec=) are unknown. We use **type** to specify such types:

```
type :sec,  '()             → Integer i    { i == @sec }'
type :sec=, '(Integer i) → Integer out { i == @sec }'
```

With these annotations, verification of mix succeeds. The verifier translates Ruby's mix to a homonymous Rosette function

```
(define (mix self t1 t2 t3)
  (set-object-@sec! self (sec t1))
  (set-object-@min! self (min t2))
  (set-object-@hour! self (hour t3))
  self)
```

where (set−**object**−x! y w) writes w to the x field of y and the field selectors sec, min, and hour are uninterpreted functions. Note that **self**, the receiver of a method call that is implicit in Ruby, turns into an explicit additional argument in the Rosette definition. Verification of mix reduces to a Rosette **verify** query that, assuming mix's arguments are valid, asserts that the result is valid as well.

2.4 Method Calls

The translation of the Ruby method call e.m(e1, .., en) is challenging for two reasons. The method definition used at runtime depends on the type of the receiver e that is not always known statically, due to dynamic dispatch. But, even if the type of e is resolved to a class A, the definition of the method A.m might not be statically available, due to metaprogramming. We address these problems using just-in-time type checking and assume-guarantee reasoning.

Just-in-Time Type Checking The translation resolves the type of the receiver e by asking RDL, the just-in-type unrefined-type checker. To type an expression e, RDL requires the user to provide type annotations for any methods and instance variables that appear inside e and at runtime tags each subexpression of e with its type (similar to Dean et al. [8]). This requirement works well under metaprogramming, since the user can also use metaprogramming to provide a type for each dynamically defined method (§ 3.2). To get the type of e, the translation simply look up its tag.

To translate the method call e.m(e1, .., en) in Rosette, we first use RDL to find the class (say A) that the receiver belongs to and then correctly dispatch the call to the method A.m. Should a receiver have a union type, we emit Rosette code which branches on the potential types of the receiver using **object** class IDs, and dispatches the appropriate method in each branch.

Assume-guarantee Reasoning Once the method being called is determined, we translate the call into Rosette. As an example, lets see the method to_sec that converts **Time** to seconds, after it calls the method incr_sec from § 2.1.

```
type '(Time t { t. is_valid }) → Integer r { 0≤r<90060 }'
def to_sec(t)
  incr_sec(t.sec) + 60 ∗ t.min + 3600 ∗ t.hour
end
```

The specification of to_sec requires the result to be less than 90060, which must be true when the hour, min and increased sec are valid.

For verification, one could just inline the definition of incr_sec, but this is not always possible, or desirable. A method's definition may not be available during verification, because the method comes from a library, is external (§ 3.1) or hasn't even been defined yet (§ 3.2). The method may also contain constructs which are difficult to verify properties of, like diverging loops. To translate such calls, we use the rely-guarantee instead of using the method's definition during verification, we approximate the method by only using it's specification, *i.e.,* the user provided refinement type for the method. To precisely reason with only method's specification we follow Dafny [14] and treat pure and impure methods differently.

Pure methods Pure methods have no side effects (including mutation) and return the same result for the same inputs (satisfying the congruence axiom). Thus pure methods can be represented using Rosette's uninterpreted functions. The method incr_sec is indeed a pure function, so we can label it as such:

```
type :incr_sec, '(Integer x {0≤x<60}) →Integer r {0≤r<60}', :pure
```

With the above **pure** label the translation of to_sec treats incr_sec as an uninterpreted function. Furthermore, it asserts that the precondition 0≤x<60 holds, and assumes that the postcondition 0≤r<60 holds, which is enough information to prove to_sec safe.

Impure methods Most Ruby methods have effects, usually mutating the fields of their inputs, and thus cannot be encoded as pure. As an example, consider incr_min, a method that, given a **Time** object, increases the second and possibly the minute fields.

```
type '(Time t {t. is_valid ∧ t.min < 60})
      → Time r {r. is_valid }', protects: {t⇒@hour}
def incr_min(t)
  if t.sec < 59 then t.sec = incr_sec(t.sec)
  else t.min += 1; t.sec = 0 end
  return t
end
```

Since the sec and min fields of the input object are mutated, incr_min cannot be marked as pure. So, translation generates a fresh value as the output for incr_min that satisfies the method's post-condition. Importantly, translation of the method call *havocs* (set to fresh symbolic values) all the fields of arguments to incr_min since they could be modified. To increase precision, we provide a **protects** label to label fields of the inputs that remain unchanged. There fields are not havoced by the translation. Note how the hour field is marked as protected.

In summary, our implementation provides three labels on method specifications, each of which uses a different rule in our translation (Figure 3). The **exact** label (Rule T-EXACT) inlines the method definition (thus, can can be used only when the definition is available), the **pure** label (Rule T-PURE1) marks effect-free, congruent methods, and the **protects** {ti ⇒ fi} label (Rule T-IMPURE1) protects the field fi of the input ti. The default (and sound) label is protecting nothing **protects** {}. Currently, the labels are trusted and checking them is left as future work.

3 Just-In-Time Verification

Next, we see how labeled refinement types as method specifications are used to verify dynamic Ruby code that uses mixins (§ 3.1) and metaprogramming (§ 3.2).

3.1 Mixins

Ruby implements mixins via its *module* system. A Ruby module is a collection of method definitions which can be added to any class that *includes* the module, which happens at runtime. Interestingly, because modules are meant to be a dynamic extension to other classes, they may refer to other methods which don't exist within their environment. Such incomplete environments pose a challenge for verification.

Consider the following method div_by_val, which is defined within a module **Arithmetic**. This example is inspired by the Ruby `Money` library that we use in our evaluations 5.

```
module Arithmetic
...
  type '(Integer dividend)→ Float quotient {{quotient==dividend/value}}'
  def div_by_val (dividend)
    dividend/value
  end
end
```

The module method div_by_val takes an input dividend and divides it by value. We wish to verify the simple assertion that the result quotient is equal to dividend divided by value. This assertion isn't entirely trivial, since RTR will automatically emit the assertion that we never divide by 0.

However, value is a call to a method which does not exist anywhere in the current environment! Rather, it is expected to be defined wherever the **Arithmetic** module is included. To proceed with verification, the programmer must provide an annotation for fractional which will be used by our translation (rules T-PURE1 or T-IMPURE1 of Figure 3) to simulate its calling. The annotation

```
type :value, '() → Float v {{ v != 0 }}', :pure
```

encodes value as an uninterpreted function allowing verification of div_by_val to succeed. Note that if value were not pure, and thus labeled with `protects`, then verification of div_by_val would be impossible, since it relies on different calls to value (in the method body and in the type refinement) with the same input returning the same output.

Optionally, our system can verify external method annotations, like div_by_val 's, once the method is actually defined. The programmer may include a type checking or verification policy for the fractional method, which will be applied when the mixin is included in the environment where the method is defined, following an assume-guarantee paradigm [12]. For instance, the **Arithmetic** module can be included in a class Money which defines value:

```
class Money
  include Arithmetic
  ...
  def value
    if @val > 0 then return @val else return 0.01 end
  end
end
```

Since **Arithmetic** is included in Money, the annotation for value will be verified upon definition of Money's value method. Thus, we *assume* value's annotation to be true while we are verifying div_by_val , then we *guarantee* the annotation is indeed valid once value is defined.

3.2 Metaprogramming

Metaprogramming in Ruby allows for editing and generating code at runtime. New methods can be defined on the fly, imposing a high challenge for static verification since properties to be checked may rely on code which does not exist yet. To achieve verification in the face of metaprogramming, we employ the just-in-time checking approach proposed in [22], in which, in addition to code, method annotations can also be generated dynamically.

We illustrate the just-in-time approach using an example from the Rails app `Boxroom`, an app for managing and sharing files in a web browser. The app defines the class **UserFile**, a Rails *model* that corresponds to a table in the database.

```
class UserFile < ActiveRecord::Base
  belongs_to :folder
  ...
  type '(Folder target) → Bool b {folder == target}'
  def move(target)
    self.folder = target
    save!
  end
end
```

Every **UserFile** is associated with a folder (another model), as declared by the method call to belongs_to. The move method updates the associated folder of a

UserFile and saves the result to the database. We annotate move to specify that the new folder associated with the **UserFile** should be the same as the input target folder.

However, neither the method folder= nor folder are statically defined anywhere. Rather, the method call to belongs_to results in the dynamic generation of both these methods. We instrument the belongs_to method so that it further generates type annotations for the generated methods, with the below code.

```
module ActiveRecord::Associations::ClassMethods
  pre(:belongs_to) do |*args|
    name = args[0].to_s
    cname = name.camelize
    type '#{name}', '() → #{cname} c', :pure
    type '#{name}=',
         '(#{cname} i) → #{cname} o {#{name} == i}',
         protects: []
    true
  end
end
```

We use **pre**, an RDL method which defines a precondition contract, to define a code block (*i.e.,* an anonymous function) which will be executed on each call to belongs_to. First, the block sets variables name and cname to be the string version of the first argument passed to belongs_to and its camelized representation, respectively. In our example, name and cname will be set to " folder " and "**Folder**", respectively. Then, we define the type annotations for the methods referred to by name and name=. Finally, we trivially return true so that the contract will always pass. In our example, the following two specifications will be generated:

```
type 'folder', '() → Folder c', :pure
type 'folder=', '(Folder i) → Folder o {folder == i}', protects: []
```

These annotations will be generated when belongs_to is invoked with the :folder argument, which happens exactly after the **UserFile** class is loaded. Thus, verification of the initial move method succeeds. Even though folder= is not labeled as pure (*i.e.,* it does not satisfy congruence and may have side effects), its postcondition exactly captures the verification requirement of move that folder == target.

In short, with just-in-time specification we can verify methods in the face of metaprogramming. We use dynamically provided information to generate expressive annotations at runtime for methods that do not even exist statically.

4 From Ruby to Rosette

In this section we formally describe the translation from Ruby to Rosette programs. We start (§ 4.1) by defining the subset of Ruby λ^{RB} that is important in our translation. We achieve the translation to Rosette by first translating to

$$
\begin{array}{rl}
\textbf{\textit{Constants}} & c ::= \texttt{nil} \mid \mathit{true}, \mathit{false}, \mid 0, 1, -1, \ldots \\
\textbf{\textit{Expressions}} & e ::= c \mid x \mid x{:=}e \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid e\ ;\ e \\
 & \quad \mid \texttt{self} \mid f \mid f{:=}e \mid e.m(\overline{e}) \mid A.\texttt{new} \mid \texttt{return}(e) \\
\textbf{\textit{Refined Types}} & t ::= \{x : A \mid e\} \\
\textbf{\textit{Program}} & P ::= \cdot \mid d, P \mid a, P \\
\textbf{\textit{Definition}} & d ::= \texttt{def}\ A.m(\overline{x : t}) :: t; l = e \\
\textbf{\textit{Annotation}} & a ::= A.m :: (\overline{x : t}) \to t\ ;\ l \\
\textbf{\textit{Labels}} & l ::= \texttt{exact} \mid \texttt{pure} \mid \texttt{protects}[\overline{x.f}]
\end{array}
$$

$x \in$ var ids, $f \in$ field ids, $m \in$ meth ids, $A \in$ class ids

Fig. 1. Syntax of the Ruby Subset λ^{RB}.

$$
\begin{array}{rl}
\textbf{\textit{Values}} & w ::= c \mid \texttt{object}(i,\ i,\ \overline{[f\ w]}) \\
\textbf{\textit{Expressions}} & u ::= w \mid x \mid x{:=}u \mid \texttt{if}\ u\ \texttt{then}\ u\ \texttt{else}\ u \mid u\ ;\ u \\
 & \quad \mid \texttt{let}\ (\overline{[x\ u]})\ \texttt{in}\ u \mid x(\overline{u}) \mid \texttt{assert}(u) \\
 & \quad \mid \texttt{assume}(u) \mid \texttt{return}(u) \mid \texttt{havoc}(x.f) \mid x.f := u \mid x.f \\
\textbf{\textit{Program}} & Q ::= \cdot \mid d, Q \mid v, Q \\
\textbf{\textit{Definition}} & d ::= \texttt{define}\ x(\overline{x}) = u \mid \texttt{define-obj}(x, i,\ i, \overline{[x\ u]}) \\
 & \quad \mid \texttt{define-sym}(x,\ A) \\
\textbf{\textit{Verification Query}} & v ::= \texttt{verify}(\overline{u} \Rightarrow u)
\end{array}
$$

$x \in$ var ids, $f \in$ field ids, $A \in$ types, $i \in$ integers

Fig. 2. Syntax of the Intermediate Language λ^{I}.

an intermediate language λ^I (§ 4.2). Then (§ 4.3), we discuss how λ^I maps to a Rosette program. Finally (§ 4.5), we use this translation to construct a verifier for Ruby programs.

4.1 Core Ruby λ^{RB} & Intermediate Representation λ^{I}

λ^{RB} Figure 1 defines λ^{RB}, a core Ruby-like language. *Constants* consist of `nil`, booleans, and integers. *Expressions* include constants, variables, conditionals, sequences, and the reserved variable `self` which refers to a method's receiver. Also included are references to an instance variable f, instance variable assignment, method calls, constructor calls A.`new` which create a new instance of class A, and return statements.

Refined types in λ^{RB}, $\{x : A \mid e\}$ include a base type A which is used to represent all types including integers, floats, booleans, *etc.*. They also include a *pure, boolean valued* refinement expression e, to denote expressions x of type A that also satisfy the predicate e. That is, the variable x can appear free in the expression e. In the interest of greater simplicity in the translation, we require that $self$ *not* appear in refinements e; however, extending the translation

to handle this is natural, and our implementation allows for it. Sometimes we simplify the trivially refined type $\{x : A \mid \texttt{true}\}$ to just A.

A *program* is a sequence of method definitions and type annotations over methods. A method definition $\texttt{def}\ A.m(x_1 : t_1, \ldots, x_n : t_n) :: t; l = \ e$ defines the method $A.m$ with arguments $x_1, \ldots, x_n$ and body e. The type specification of the definition is a dependent function type: each argument binder x_i can appear inside the argument refinement types t_j for all $1 <= j <= n$, and can also appear in the refinement of the result type t. A method type annotation $A.m :: (\overline{x : t}) \to t\ ;\ l$ binds the method named $A.m$ with the dependent function type $(\overline{x : t}) \to t$. We include method annotations independent of method definitions because annotations may be used when a method's code is not available, *e.g.,* in the cases of library methods, mixins, or metaprogramming.

A *label* l annotates both method definitions and annotations to direct the method's translation into Rosette. The label `exact` states that a called method will be exactly translated by using the translation of the body of the method. Since method type annotations do not have a body, they cannot be assigned the `exact` label. The `pure` label indicates that a method is pure and thus can be translated using an uninterpreted function. Finally, the $\texttt{protects}[\overline{x.f}]$ label is used when a method is impure. This means that the method may modify its inputs. The list of fields of method arguments given by $\overline{x.f}$ captures all the argument fields which the method does *not* modify, information which we can use when translating the method call. Thus, our default assumption (when the list given to `protects` is empty) is imprecise but sound because it assumes all argument fields are modified.

λ^I Figure 2 defines λ^I, a core verification-oriented language that easily translates (§ 4.3) to Rosette. λ^I maps objects to functional structures. Concretely, λ^{RB} objects map to a special **object** struct type, and λ^I provides primitives for creating, altering, and referencing instances of this type. *Values* consist of *Constants* c (defined identically as in λ^{RB}) and $\texttt{object}(i_1,\ i_2,\ [x_1\ v_1] \ldots [x_n\ v_n])$, an instantiation of an **object** struct with class ID i_1, object ID i_2, and where each field x_i of the **object** is bound to value v_i. *Expressions* consist of `let` bindings $(\texttt{let}\ (\overline{[x_i\ u_i]})\ \texttt{in}\ u)$ where each x_i may appear free in u_j if $i < j$, function calls, **assert**, **assume**, and **return** statements. They also include $\texttt{havoc}(x.f)$, which mutates x's field f to a fresh symbolic value. Finally, they include field assignment $x.f := u$ and field reads $x.f$, which respectively set and get the field f of the `object` given by x.

A *program* is a series of definitions and verification queries. A *definition* is a function definition, an object definition $\texttt{define-obj}(x, i_1,\ i_2, [x_1\ u_1] \ldots [x_n\ u_n])$, where x is bound to a new **object** with class ID i_1, object ID i_2, and field x_i bound to u_i, or a symbolic object definition $\texttt{define-sym}(x,\ A)$, where x is bound to a new **object** with symbolic fields defined depending on the type A. Finally, a verification query $\texttt{verify}(\overline{u} \Rightarrow u)$ checks the validity of u assuming $\overline{u}$.

Expression Translation $\boxed{e \rightsquigarrow u}$

$$\frac{}{c \rightsquigarrow c}\ \text{T-Const} \qquad \frac{}{x \rightsquigarrow x}\ \text{T-Var} \qquad \frac{e_1 \rightsquigarrow u_1 \quad e_2 \rightsquigarrow u_2}{e_1\ ;\ e_2 \rightsquigarrow u_1\ ;\ u_2}\ \text{T-Seq}$$

$$\frac{e_1 \rightsquigarrow u_1 \quad e_2 \rightsquigarrow u_2 \quad e_3 \rightsquigarrow u_3}{\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \rightsquigarrow \texttt{if}\ u_1\ \texttt{then}\ u_2\ \texttt{else}\ u_3}\ \text{T-If} \qquad \frac{}{\texttt{self} \rightsquigarrow self}\ \text{T-Self}$$

$$\frac{f \in \mathcal{F}}{f \rightsquigarrow self.f}\ \text{T-Inst} \qquad \frac{f \in \mathcal{F} \quad e \rightsquigarrow u}{f{:=}e \rightsquigarrow self.f := u}\ \text{T-InstAssn}$$

$$\frac{e \rightsquigarrow u}{x{:=}e \rightsquigarrow x{:=}u}\ \text{T-VarAssn} \qquad \frac{e \rightsquigarrow u}{\texttt{return}(e) \rightsquigarrow \texttt{return}(u)}\ \text{T-Ret}$$

$$\frac{\texttt{classId}(A) = i_c \quad freshID(i_o) \quad \overline{f_i} \in \mathcal{F}}{A.\texttt{new} \rightsquigarrow \texttt{object}(i_c,\ i_o,\ \overline{[f_i\ \texttt{nil}]})}\ \text{T-New}$$

$$\frac{\begin{array}{c}\texttt{typeOf}(e_F) = A \quad \texttt{exact} = \texttt{labelOf}(A.m) \\ A\_m \in \mathcal{M} \qquad e_F \rightsquigarrow u_F \qquad e_i \rightsquigarrow u_i\end{array}}{e_F.m(\overline{e}) \rightsquigarrow A\_m(u_F, \overline{u})}\ \text{T-Exact}$$

$$\frac{\begin{array}{c}\texttt{typeOf}(e_F) = A \qquad \texttt{pure} = \texttt{labelOf}(A.m) \\ A\_m \in \mathcal{U} \qquad freshVar(a) \\ \texttt{specOf}(A.m) = (x_{in} : \{x_{in} : A_{in}\ \mid\ e_{ri}\}) \to \{x_{ro} : A\ \mid\ e_{ro}\} \\ e_F \rightsquigarrow u_F \qquad e_{in} \rightsquigarrow u_{in} \qquad e_{ri} \rightsquigarrow u_{ri} \qquad e_{ro} \rightsquigarrow u_{ro}\end{array}}{\begin{array}{l}e_F.m(e_{in}) \rightsquigarrow \texttt{let}\ ([a_i\ u_{in}][x\ A\_m(u_F, a)])\ \texttt{in} \\ \quad \texttt{assert}(u_{ri}[x_{in} \mapsto a])\ ;\ \texttt{assume}(u_{ro}[x_{in} \mapsto a][x_{ro} \mapsto x])\ ;\ x\end{array}}\ \text{T-Pure1}$$

$$\frac{\begin{array}{c}\texttt{typeOf}(e_F) = A \quad \texttt{protects}[\overline{x_{in}.f}, \overline{self.f}] = \texttt{labelOf}(A.m) \\ \texttt{specOf}(A.m) = (x_{in} : \{x_{in} : A_{in}\ \mid\ e_{ri}\}) \to \{x_{ro} : A\ \mid\ e_{ro}\} \\ freshVar(x) \qquad freshVar(a) \\ e_F \rightsquigarrow u_F \qquad e_{in} \rightsquigarrow u_{in} \qquad e_{ri} \rightsquigarrow u_{ri} \qquad e_{ro} \rightsquigarrow u_{ro}\end{array}}{\begin{array}{l}e_F.m(\overline{e}) \rightsquigarrow \texttt{let}\ ([a\ u_{in}])\ \texttt{in} \\ \quad \texttt{assert}(u_{ri}[x_{in} \mapsto a])\ ;\ \texttt{define-sym}(x,\ A)\ ;\ \texttt{havoc}((x_{in}.f_i \notin \overline{x_{in}.f})[x_{in} \mapsto a])\ ; \\ \quad \texttt{havoc}(self.f_i \notin \overline{self.f})\ ;\ \texttt{assume}(u_{ro}[x_{in} \mapsto a][x_{ro} \mapsto x])\ ;\ x\end{array}}\ \text{T-Impure1}$$

Program Translation $\boxed{P \rightsquigarrow Q}$

$$\frac{}{\cdot \rightsquigarrow \cdot}\ \text{T-Emp} \qquad \frac{P \rightsquigarrow Q}{A.m :: (\overline{x : t}) \to t\ ;\ l, P \rightsquigarrow Q}\ \text{T-Ann}$$

$$\frac{\begin{array}{c}t_i = \{x_i : A_i\ \mid\ e_i\} \quad t = \{x : A_o\ \mid\ e_o\} \\ e \rightsquigarrow u \quad e_i \rightsquigarrow u_i \quad e_o \rightsquigarrow u_o \quad P \rightsquigarrow Q\end{array}}{\texttt{def}\ A.m(\overline{x : t}) :: t; l = \ e, P \rightsquigarrow \begin{array}{l}\texttt{define}\ A\_m(self, \overline{x_i}) = u; \\ \texttt{define-sym}(self,\ A); \\ \overline{\texttt{define-sym}(x_i,\ A_i)}; \\ \texttt{verify}(\overline{u_i} \Rightarrow u_o)\ ;\ Q\end{array}}\ \text{T-Def}$$

Fig. 3. Translation from λ^{RB} to λ^{I}. We mix the notation $\overline{e_i} \equiv \overline{e} \equiv e_1, \ldots, e_n$. For brevity, rules T-Pure1 and T-Impure1 only handle method calls with one argument.

4.2 From λ^{RB} to λ^{I}

Figure 3 defines the translation function $e \rightsquigarrow u$ that maps expressions (and programs) from λ^{RB} to λ^{I}.

Global States The translation uses sets $\mathcal{M}, \mathcal{U}$, and $\mathcal{F}$, to ensure all the methods, uninterpreted functions, and fields are well-defined in the generated λ^{I} term:

$$\mathcal{M} ::= A_1.m_1, \ldots, A_n.m_n \quad \mathcal{U} ::= A_1.m_1, \ldots, A_n.m_n \quad \mathcal{F} ::= f_1, \ldots, f_n$$

We use set membership $x \in \mathcal{X}$ to check if x is a member of $\mathcal{X}$. The treatment of these sets is bidirectional: the translation rules assume that these sets are properly guessed, but can be also use to construct the sets.

Expressions The rules T-Const and T-Var are identity while the rules T-If, T-Seq, T-Ret, and T-VarAssn are trivially inductively defined. The rule T-Self translates `self` into the special *variable* named $self$ in λ^{I}. The $self$ variable is always in scope, since each λ^{RB} method translates to a λ^{I} function with an explicit first argument named $self$. The rules T-Inst and T-InstAssn translate a reference from and an assignment to the instance variable f, to a respective reading from and writing to the field f of the variable $self$. Moreover, both the rules assume the field f to be in global field state $\mathcal{F}$. The rule T-New translates from a constructor call A.`new` to an `object` instance. The `classId`(A) function in the precondition of this rule returns the class ID of A. The $freshID(i_o)$ predicate ensures the new `object` instance has a fresh object ID. Each field of the new `object` is initially bound to `nil`.

Method Calls To translate the λ^{RB} method call $e_F.m(\overline{e})$ we first use use the function `typeOf`(e_F) to type e_F via RDL type checking [22]. If e_F is of type A we split cases of the method call translation based on the value of `labelOf`$(A.m)$, the label that was specified when the annotation for $A.m$ was declared.

The rule T-Exact is used when the label is `exact`. Then, the receiver e_F is translated to u_F which becomes the first (*i.e.,* the $self$) argument of the function call to $A.m$. Finally, $A.m$ is assumed to be in the global method name set $\mathcal{M}$ since it belongs to the transitive closure of the translation.

We note that for the sake of clarity, in the T-Pure1 and T-Impure1 rules, we assume that the method $A.m$ takes just one argument; the rules can be extended in the natural way to account for more arguments. The rule T-Pure1 is used when the label is `pure`. When the receiver is labeled as a pure function, its translation is an invocation to the uninterpreted function $A.m$. So, $A.m$ is assumed to be in the global set of uninterpreted functions $\mathcal{U}$. To ensure that the argument(s) will be evaluated exactly once, we convert the translated expression into A-normal form [25]. Moreover, the specification of the function is checked. That is, using `specOf`$(A.m)$ to get the function's refinement type specification, we assert the function's precondition(s) and assume the postcondition.

If a method is labeled with `protects`$[\overline{x.f}]$ then the rule T-Impure1 is applied. Since the method is not pure, it cannot be encoded as an uninterpreted function. Instead, we locally define a new symbolic object as the return value, and we havoc the fields of all arguments (including $self$) which are *not* in the `protects` label, thereby assigning these fields to new symbolic values. Since we do not translate the called method at all, no global state assumptions are made.

Programs Finally, we use the translation relation to translate programs from λ^{RB} to λ^I, *i.e.,* $P \rightsquigarrow Q$. The rule T-Ann does not translate type annotations.

The rule T-Def translates a method definition for $A.m$ to the function definition $A.m$ that takes the additional first argument $self$. The rule also considers the declared type of $A.m$ and instantiates a symbolic object for every input argument. Finally, all refinements from the inputs and output of the method type are translated and the derived verification query is made.

4.3 From λ^I to Rosette

The translation from λ^I to Rosette is straightforward, since λ^I consists of Rosette extended with some macros to encode Ruby-verification specific operators, like `define-obj` and `return`. In fact, in the implementation of the translation (§ 5) we used Racket's macro expansion system to achieve this final transformation.

Handling objects λ^I contains multiple constructs for defining and altering objects, which will be expanded in Rosette to perform the associated operations over **object** structs. `define-obj`$(x, i,\ i, [x_1\ u_1] \ldots [x_n\ u_n])$ is a macro that binds x to a new **object** with class ID e, object ID i and where each field x_i of x is bound to e_i. `define-sym`$(x,\ A)$ creates a new symbolic object bound to x. If A is one of Rosette's solvable types, x is simply an **object** which boxes a symbolic value of type A. Otherwise, a new **object** is created with fields instantiated with symbolic objects of the appropriate type. Finally, `havoc`$(x.f)$ is expanded to mutate all the f field of x to a new symbolic object of the appropriate type.

Control Flow Macro expansion is used to translate both `return` and `assume` in Rosette. In order to encode **return**, every function definition in λ^I is expanded to keep track of a local variable ret, which is initialized to a special undefined value, and is returned at the end of the function. Every statement `return`(e) is transformed to update the value of ret to e. Then, every expression u in a function is expanded to `unless-done`(u). unless−done is a function that checks the value of ret. If ret is undefined (*i.e.,* nothing has been returned yet) then we proceed with executing u. Otherwise (i.e., something has been returned) u is not executed.

We used the encoding of `return` to encode more operators. For example, `assume` is encoded in Rosette as a macro that returns a special fail value when assumptions do not hold. The verification query then needs to be updated with the condition that fail is not returned. Similarly, in the implementation, we used macro expansion to encode and propagate exceptions.

4.4 Primitive Types

The core language of λ^{RB} provides primitives for functions, assignments, control flow, *etc.*, but does not provide the language required to encode interesting verification properties, that for example reason about booleans and numbers. On the other hand, Rosette is a verification oriented language with special support for common theories over built-in datatypes, including booleans, numeric types, and vectors. To bridge this gap, we unbox certain Ruby expressions, encoded as objects in λ^{RB}, into Rosette's built-in datatypes.

Equality and Booleans To precisely reason about equality, we encode Ruby's == method over arbitrary objects using Rosette's equality operator equal? to check equality of objects' IDs. We encode Ruby's booleans, that is expressions of type TrueClass union FalseClass class and operators over such expressions, as Rosette's respective boolean data and operations.

Integers and Floats By default, we encode Ruby's infinite-precision **Integer** and **Float** objects as Rosette's built-in infinite-precision integer and real datatypes, respectively. The infinite-precision encoding is efficient and precise, but it may result in undecidable queries involving non-linear arithmetic or loops, for example. To reason in such cases we provide the alternative encoding of Ruby's integers as Rosette's built-in finite sized bitvectors that come equipped with arithmetic operators.

Arrays Finally, we provide a special encoding for Ruby's arrays, since arrays in Ruby are commonly used both for storing arbitrarily large random-access data, and also for representing mixed-type tuples, stacks, queues, *etc.*. We encode Ruby's arrays as a Rosette struct composed of a fixed-size vector and an integer that represents the current size of the Ruby array. We chose to explicitly preserve the vector's size an a Rosette integer to get more efficient and precise reasoning when we index or loop over vectors.

4.5 Verification of λ^{RB}

We define a verification algorithm RTR that, given a λ^{RB} program P checks if it is *safe*, *i.e.,* all the definitions satisfy the specifications. The pseudo-code for this algorithm is shown below. First, it defines an **object** struct in Rosette containing one field for each member of $\mathcal{F}$, and it defines an uninterpreted function for each method in $\mathcal{U}$. Then, it translates P to the λ^I program Q via $P \rightsquigarrow Q$ (§ 4.2). Next, it translates Q to a Rosette program via macro expansion (§ 4.3). Finally, it runs the Rosette program. If the Rosette program is *valid*, *i.e.,* all the `verify` queries are valid, then the initial program is *safe*.

for $(f \in \mathcal{F})$: add field f to **object struct**
for $(u \in \mathcal{U})$: define uninterpreted function u
P := definitions $m \in \mathcal{M}$
$P \rightsquigarrow Q$

if (*valid*(Rosette (Q))) **return** SAFE
else return UNSAFE

We conclude this section with a discussion of the RTR verifier.

RTR is Partial There exist expressions of λ^{RB} that fail to translate into a λ^{I} expression. The translation requires at each method call $e_F.m(\overline{e})$ the receiver has a class type A (*i.e.,* $\texttt{specOf}(e_F) = A$). There are three cases when this requirement fails 1) e_F has a union type, 2) e_F is `nil`, 3) type checking fails and so e_F has no type. In our implementation, we extend the translation to the first two cases. Handling for 1) is outlined in § 2.4. In case 2), we treat the receiver as `self`, matching the Ruby semantics. Case 3) can be caused by either a type error in the program or a lack of typing information for the type checker. In both cases translation cannot proceed.

RTR may Diverge The translation to Rosette always terminates. The transitive method closure $\mathcal{M}$, which is iterated over until all of its members have been translated, is finite since any program being translated will be finite. Additionally, in each case of the translation's inductive rules, an expression or program is translated to a syntactically smaller result. Finally, all the helper functions (including the type checking $\texttt{specOf}(\cdot)$) do terminate.

Yet, verification may diverge, as the translated Rosette program may diverge. Specifications can encode arbitrary expressions, thus it is possible to encode undecidable verification queries. Consider the following trivial Rosette program in which we attempt to verify an assertion over a recursive method:

```
(define (rec x) (rec x))
(define-symbolic b boolean?)
(verify (rec b))
```

Rosette will attempt to symbolically evaluate this program, and as a result will not terminate.

RTR is Incomplete Verification is incomplete and its precision relies on the precision of the specifications. For instance, if a pure method $A.m$ is marked as impure, the verifier will not prove the congruence property $A.m(x) = A.m(x)$.

RTR is Sound If the verifier decides that the input program is safe, the all definitions satisfy their specifications, assuming that 1) all the refinements are pure boolean expressions and 2) all the labels are sounds. The assumption 1) is required since verification under diverging (let alone effectful) specifications is difficult [33] The assumption 2) is required since our translation encodes pure methods as uninterpreted functions while for the impure methods it only havocs the unprotected arguments. Checking these assumptions is left as future work.

5 Evaluation

We implemented our translation in RTR, an extension of RDL [22] with specifications in the form of refinement types. Table 2 summarizes the evaluation of RTR.

Benchmarks To evaluate our technique, we used five popular Ruby libraries.

- `Money` performs currency conversions over monetary quantities,
- `BusinessTime` performs time calculations in business hours and days,
- `Unitwise` performs various unit conversions,
- `Geokit` performs calculations over locations on Earth, and
- `Boxroom` is a Ruby on Rails app for sharing files in a web browser.

For verification we forked the original Ruby libraries and provided manually written method specifications in the form of refinement types. The forked repositories can be found in Table 1.

We chose these libraries because they combine Ruby-specific features challenging in verification, like metaprogramming and mixins, with arithmetic heavy operations. In all libraries we verify both the functional correctness of arithmetic operations, *e.g.,* no division-by-zero, the absolute value of a number should not be negative, and data specific arithmetic invariants, *e.g.,* integers representing months should always be in the range from `1` to `12` and a `data` value added to an aggregate should always fall between maintained `@min` and `@max` fields.

Table 1. Source code for Verified Libraries forked from their original versions.

`Money`	`https://github.com/mckaz/RubyMoney`
`BusinessTime`	`https://github.com/mckaz/bussiness_time_verify`
`Unitwise`	`https://github.com/mckaz/unitwise_verify`
`Geokit`	`https://github.com/mckaz/geokit_verify`
`Boxroom`	`https://github.com/mckaz/boxroom-verify`

Table 2. Evaluation of RTR. **Methods** is the number of methods verified. **Ruby LoC** is the range of the LoC per method of the verified methods and **Rosette LoC** is the range of the LoC per method of the Rosette translation. **Verification Time** is the median and range of the total verification time in seconds for 11 runs. **Spec** is the number of total refinement type specifications required for verification. Experiments were conducted on a 2014 Macbook Pro with a 3 GHz Intel Core i7 processor and 16 GB memory.

App	Methods	Ruby LoC	Rosette LoC	Verification Time		Spec
				Time(s)	SIQR	
`Money`	7	43 [5, 11]	197 [24, 40]	23.24	0.29	10
`BusinessTime`	10	76 [5, 12]	337 [26, 58]	34.00	0.16	24
`Unitwise`	6	24 [4, 4]	153 [22, 27]	19.51	0.21	6
`Geokit`	6	59 [5, 26]	246 [26, 68]	22.63	0.07	21
`Boxroom`	1	12 [12, 12]	34 [34, 34]	3.28	0.04	3

Quantitative Evaluation Table 2 summarizes our evaluation in numbers. For each application we list the number of verified **Methods**. RTR acts at a method-level

granularity, that is, the programmer can specify exactly which methods should be verified, and when they should be verified (e.g., immediately when it is defined, prior to executing the method when it is called, or at a separate customized time). Yet, verified methods co-exist with (and can even call) the non-verified, modularly used methods. We only verified methods with interesting arithmetic properties.

The **Ruby LoC** and **Rosette LoC** columns give the sizes of the verified Ruby programs and translated Rosette programs, respectively. These metrics show the sum total lines of code (LoC) over all verified Ruby programs and resulting Rosette programs, followed by the range of LoC over these programs. For example, for the `Money` app, the sum total Ruby LoC over all 7 methods verified was 43, with the smallest method consisting of 5 LoC, and the largest consisting of 11. For each method verified, we generate a separate Rosette program, and give the sizes of these programs and their sum in the **Rosette LoC** column.

To present the size of the verified methods we show the **Ruby LoC** metric that represents the range of lines of code over all verified method. **Rosette LoC** represents the range of lines of code of the methods' translation to Rosette programs. Unsurprisingly, the LoC of the Rosette generated program increases with the size of the source Ruby program.

We present the median (**Time(s)**) and semi-interquartile range (**SIQR**) of the **verification time** required to verify all methods for an application over 11 runs. For each application, the **SIQR** fell under 1 second, indicating relatively little variance in the time taken to verify all methods over 11 runs. From our experiments we conclude that verification time depends on both the size of the verified methods and on the complexity of the specifications.

Finally, Table 2 lists the number of type **specifications** required to verify an application's methods. These are comprised of method type annotations, including the annotations for the verified methods themselves, and variable type annotations for instance variables. Not listed, but nevertheless crucial to verification, are the number of type annotations from Ruby's standard and core libraries that are used; RDL includes an extensive list of annotations for Ruby library methods, so we did not need to write these manually for verification.

Notably, there is a high level of variation in the number of annotations required for each application. For example, in `Unitwise` we verified 6 methods and only required 6 annotations (one for each verified method), while in `Geokit` we verified 6 methods and required 21 annotations. The large variation results from the differing natures of the methods being verified. For each instance variable used in a method, and for each additional (non-standard/core library) method called, the programmer must provide an additional annotation. These requirements can account for large variation in number of annotations needed.

5.1 Case Study

Next we illustrate the RTR verification process by presenting the exact steps required to specify and check the properties of a method from an existing Ruby library. For this process, we chose to verify the << method of the `Aggregate`

library [1], a Ruby library for aggregating and performing statistical computations over some numeric data. The method (<<) takes one input, data, and adds it to the aggregate by updating (1) the minimum @min and maximum @max of the aggregate, (2) the count @count, sum @sum, and sum of squares @sum2 of the aggregate, and finally (3) the correct bucket in @buckets.

```
def << data
  # Update min/max
  if 0 == @count
    @min = data
    @max = data
  else
    @max = data if data > @max
    @min = data if data < @min
  end
  # Update the running info
  @count += 1
  @sum  += data
  @sum2 += (data * data)
  # Update the bucket
  @buckets[to_index(data)] += 1 unless outlier?(data)
end
```

We specify functional correctness of the method << by providing a refinement type specification that declares that after the method is executed, the input data will fall between @min and @max.

```
type :<<,
     '(Integer data) → Integer { @min ≤ data ≤ @max }', verify: :bind
```

The type specification is stored in a global table, and includes a user specified verification *alias*, in this case :bind. To verify the specification all we need to do is load the library and call the verifier with the method's verification alias:

```
rdl_do_verify :bind
```

Verification proceeds in three steps

- first use RDL to type check the the basic types of the method,
- then translate the method to Rosette (using the translation of 4), and
- finally run the Rosette program to check the validity of the specification.

At the current state, verification will fail in the first step, due to a lack of type information. Type checking will fail with the error

```
error: no type for instance variable '@count'
```

To fix this error, the user needs to provide the correct types for the instance variables using the below type annotations.

[1] The code for the `Aggregate` library is taken from `https://github.com/josephruscio/aggregate`

```
var_type :@count, 'Integer'
var_type :@min, :@max, :@sum, :@sum2, 'Float'
var_type :@buckets, 'Array<Integer>'
```

The << method also calls two methods which are *not* from Ruby's standard and core libraries: to_index, which takes a numeric input and determines the index of the bucket the input falls in, and outlier, which determines whether the given data is an outlier based on provided specifications from the programmer. These methods provide challenging obstacles to verification. For example, the to_index method makes use of non-linear arithmetic in the form of logarithms, and loops over arrays. Yet, neither of the calls to_index or outlier? should affect verification of the specification of <<. So, it suffices to provide type annotations with a `pure` label, indicating we want to use uninterpreted functions to represent them:

```
type : outlier ?, '(Float i) → Bool b', :pure
type :to_index, '(Float i) → Integer out', :pure
```

Given these annotations, the verifier has enough information to prove the post-condition on <<, and it will return the message to the user:

```
Aggregate instance method << is safe.
```

When verification fails, an unsafe message is provided, combined with a counterexample consisting of bindings to symbolic values which causes the postcondition to fail. For instance, if the programmer incorrectly specified that data is less than the @min, *i.e.,*

```
type :<<, '(Integer data) → Integer { data < @min }'
```

Then RTR would return the following message:

```
Aggregate instance method << is unsafe.
Counterexample: (model [real_data 0]
                       [real_@min 0] ...)
```

This gives a binding to symbolic values in the translated Rosette program which would cause the specification to fail. We only show the bindings relevant to the specification here: when real_data and real_@min, the symbolic values corresponding to data and @min respectively, are both 0, the specification fails.

6 Related Work

Verification for Ruby There is some prior work on verifying expressive properties of Ruby programs, with a particular focus on Ruby on Rails web applications. *Space* [18] aims to detect security bugs in Rails apps by using symbolic execution to generate a model of data exposures in the app, and comparing this model against a catalog of common access control patterns. Space reports a possible bug when no matches are found. In contrast to our own work, Space does not let the programmer specify their own properties to be verified, and it only applies to Rails apps, not general Ruby programs.

Symbolic model extraction [6] is proposed and implemented for Rails apps. This approach instruments code such that its execution outputs a model of

the code. Rails apps are then run to extract models, allowing for dynamically generated code (*i.e.*,, metaprogramming) to be modeled as well, and generated models are then used to verify data integrity and access control properties. This approach also does not allow programmers to specify their own properties to be verified, and is only applied to Rails apps.

Rubicon [17] introduces bounded verification for Rails apps. It allows programmers to write specifications using a domain-specific language that looks similar to tests for Rails apps, but includes the ability to quantify over objects. It then uses symbolic execution generate logical constraints. *Rubyx* [7] likewise allows programmers to write their own specifications over Rails apps, and uses symbolic execution to verify these specifications. Both of these systems are specific to the Rails framework, and do not generalize verification to the Ruby language.

Rosette Rosette has been used to help establish the security and reliability of multiple real-world software systems. Pernsteiner et al. [19] uses Rosette to build a verifier to study the safety of the software on a radiotherapy machine. The verifier was used as one piece in a compositional analysis of the correctness of the machine's software, hardware, and physical components, and was ultimately able to find a number of safety-critical flaws in the radiotherapy machines.

Bagpipe [34] also builds a verifier upon Rosette in order to analyze the routing protocols used by Internet Service Providers (ISPs). It was able to discover a number of policy violations existing in configurations used by ISPs. The success of Rosette in these domains indicates its efficacy in establishing the security and reliability of software systems.

Types For Dynamic Languages There have been a number of efforts to bring type systems to dynamic languages including Python [2, 4], Racket [28, 29], and JavaScript [3, 15, 27], among others. However, none of these systems use the just-in-time approach of both collecting and generating type annotations at runtime, and so to our knowledge, they all are limited in their expressivity in the face of highly dynamic features like metaprogramming.

Some systems have been developed to introduce refinement types to scripting and dynamic languages as well. *Refined TypeScript* (RSC) [32] introduces refinement types to TypeScript[5, 21], a superset of JavaScript that includes optional static typing. To our knowledge, this is the only other effort to date to bring refinement types to a scripting language. RSC additionally uses the framework of Liquid Types [23] to achieve refinement inference. Refinement types have been introduced [**?**] to Typed Racket as well. While both of these efforts introduce refinement types to dynamic languages, both attempt to check these types *at compile time*. In contrast, by using the just-in-time approach, RTR is able to generate and verify expressive specifications in the face of highly dynamic features such as metaprogramming. Furthermore, by including method annotation labels describing the purity or side effects of a method, RTR allows for more expressive annotations to be used in modular verification.

The approach of *hybrid type checking* [9] allows for arbitrary predicates in refinement types, and proposes statically verifying refinement types whenever possible, but inserting dynamic checks when static verification is not possible. This is in some ways similar to RTR, since we allow optionally treating refinement types as dynamically checked contracts. However, our approach of performing not just dynamic checks but also static verification *at runtime* allows us to verify refinement types in the face of metaprogramming, and to achieve assume-guarantee reasoning in the partial environments of mixins.

General Purpose Verification Dafny [14] is an object-oriented language with built-in constructs for high-level specification and verification. While it does not explicitly include refinement types, the ability to specify a method's type and pre- and postconditions effectively achieves the same level of expressiveness. Dafny also performs modular verification by using a method's pre- and postconditions and labels indicating its purity or arguments mutated, an approach we largely emulate here. However, unlike Dafny, RTR leaves this modular treatment of methods as an option for the programmer. Furthermore, Dafny does not include features such as mixins and metaprogramming, which RTR is able to account for.

7 Conclusion and Future Work

We formalized and implemented RTR, a refinement type checker for Ruby programs using the just-in-time checking technique of RDL. Verification at runtime naturally adjusts standard refinement types to handle Ruby's dynamic features, such as metaprogramming and mixins. To evaluate our technique we used RTR to verify commonly used Ruby and `Ruby on Rails` applications simply by adding type annotations and no code adjustments.

Our work opens new directions for useful Ruby verification. Currently the verifier trusts annotation labels that claim purity and immutability. Verification of the labels *per se* would provide better insight into Ruby methods. Next, (refinement) type inference is a feature missing from our system, crucial for usability. Our plan is to investigate how standard inference techniques, like Hindley-Milner and liquid typing [23], adjust to just-in-time typing. We also plan to extend the expressiveness of the system by adding support for loop invariants, dynamically defined instance variables, and polymorphic method calls, among other Ruby constructs. Finally, as Ruby is commonly used in the `Ruby on Rails` framework, we plan to extend RTR with modeling for web-specific constructs such as access control protocols and database operations in order to further support verification in the domain of web applications.

Bibliography

[1] Aguirre, A., Barthe, G., Gaboardi, M., Garg, D., Strub, P.Y.: A relational logic for higher-order programs (2017)

[2] Ancona, D., Ancona, M., Cuni, A., Matsakis, N.D.: Rpython: A step towards reconciling dynamically and statically typed oo languages. In: Proceedings of the 2007 Symposium on Dynamic Languages. pp. 53–64. DLS '07, ACM, New York, NY, USA (2007), http://doi.acm.org/10.1145/1297081.1297091

[3] Anderson, C., Giannini, P., Drossopoulou, S.: Towards type inference for javascript. In: Proceedings of the 19th European Conference on Object-Oriented Programming. pp. 428–452. ECOOP'05, Springer-Verlag, Berlin, Heidelberg (2005), http://dx.doi.org/10.1007/11531142_19

[4] Aycock, J.: Aggressive type inference. International Python Conference (2000)

[5] Bierman, G., Abadi, M., Torgersen, M.: Understanding typescript. In: Proceedings of the 28th European Conference on ECOOP 2014 — Object-Oriented Programming - Volume 8586. pp. 257–281. Springer-Verlag New York, Inc., New York, NY, USA (2014), http://dx.doi.org/10.1007/978-3-662-44202-9_11

[6] Bocić, I., Bultan, T.: Symbolic model extraction for web application verification. In: Proceedings of the 39th International Conference on Software Engineering. pp. 724–734. ICSE '17, IEEE Press, Piscataway, NJ, USA (2017), https://doi.org/10.1109/ICSE.2017.72

[7] Chaudhuri, A., Foster, J.S.: Symbolic security analysis of ruby-on-rails web applications. In: Proceedings of the 17th ACM Conference on Computer and Communications Security. pp. 585–594. CCS '10, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1866307.1866373

[8] Dean, J., Grove, D., Chambers, C.: Optimization of object-oriented programs using static class hierarchy analysis. In: Proceedings of the 9th European Conference on Object-Oriented Programming. pp. 77–101. ECOOP '95, Springer-Verlag, London, UK, UK (1995), http://dl.acm.org/citation.cfm?id=646153.679523

[9] Flanagan, C.: Hybrid type checking. SIGPLAN Not. 41(1), 245–256 (Jan 2006), http://doi.acm.org/10.1145/1111320.1111059

[10] Freeman, T., Pfenning, F.: Refinement types for ML (1991)

[11] Jeon, J., Qiu, X., Foster, J.S., Solar-Lezama, A.: Jsketch: Sketching for java. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. pp. 934–937. ESEC/FSE 2015, ACM, New York, NY, USA (2015), http://doi.acm.org/10.1145/2786805.2803189

[12] Jones, C.: Specification and design of (parallel) programs. 83, 321–332 (01 1983)

[13] Kent, A.M., Kempe, D., Tobin-Hochstadt, S.: Occurrence typing modulo theories. PLDI (2016)

[14] Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning. pp. 348–370. LPAR'10, Springer-Verlag, Berlin, Heidelberg (2010), http://dl.acm.org/citation.cfm?id=1939141.1939161

[15] Lerner, B.S., Politz, J.G., Guha, A., Krishnamurthi, S.: Tejas: Retrofitting type systems for javascript. SIGPLAN Not. 49(2), 1–16 (Oct 2013), http://doi.acm.org/10.1145/2578856.2508170

[16] de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver, pp. 337–340. Springer Berlin Heidelberg, Berlin, Heidelberg (2008), https://doi.org/10.1007/978-3-540-78800-3_24

[17] Near, J.P., Jackson, D.: Rubicon: Bounded verification of web applications. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. pp. 60:1–60:11. FSE '12, ACM, New York, NY, USA (2012), http://doi.acm.org/10.1145/2393596.2393667

[18] Near, J.P., Jackson, D.: Finding security bugs in web applications using a catalog of access control patterns. In: Proceedings of the 38th International Conference on Software Engineering. pp. 947–958. ICSE '16, ACM, New York, NY, USA (2016), http://doi.acm.org/10.1145/2884781.2884836

[19] Pernsteiner, S., Loncaric, C., Torlak, E., Tatlock, Z., Wang, X., Ernst, M.D., Jacky, J.: Investigating Safety of a Radiotherapy Machine Using System Models with Pluggable Checkers, pp. 23–41. Springer International Publishing, Cham (2016), https://doi.org/10.1007/978-3-319-41540-6_2

[20] Protzenko, J., Zinzindohoué, J.K., Rastogi, A., Ramananandro, T., Wang, P., Zanella-Béguelin, S., Delignat-Lavaud, A., Hriţcu, C., Bhargavan, K., Fournet, C., Swamy, N.: Verified low-level programming embedded in f* (2017)

[21] Rastogi, A., Swamy, N., Fournet, C., Bierman, G., Vekris, P.: Safe & efficient gradual typing for typescript. SIGPLAN Not. 50(1), 167–180 (Jan 2015), http://doi.acm.org/10.1145/2775051.2676971

[22] Ren, B.M., Foster, J.S.: Just-in-time static type checking for dynamic languages. PLDI (2016)

[23] Rondon, P.M., Kawaguci, M., Jhala, R.: Liquid types. PLDI (2008)

[24] Rushby, J., Owre, S., Shankar, N.: Subtypes for specifications: Predicate subtyping in pvs. IEEE Trans. Softw. Eng. (1998)

[25] Sabry, A., Felleisen, M.: Reasoning about programs in continuation-passing style. In: Proceedings of the 1992 ACM Conference on LISP and Functional Programming. pp. 288–298. LFP '92, ACM, New York, NY, USA (1992), http://doi.acm.org/10.1145/141471.141563

[26] Singh, R., Gulwani, S., Solar-Lezama, A.: Automated feedback generation for introductory programming assignments. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 15–26. PLDI '13, ACM, New York, NY, USA (2013), http://doi.acm.org/10.1145/2491956.2462195

[27] Thiemann, P.: Towards a type system for analyzing javascript programs. In: Proceedings of the 14th European Conference on Programming Languages and Systems. pp. 408–422. ESOP'05, Springer-Verlag, Berlin, Heidelberg (2005), http://dx.doi.org/10.1007/978-3-540-31987-0_28

[28] Tobin-Hochstadt, S., Felleisen, M.: Interlanguage migration: From scripts to programs. In: Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications. pp. 964–974. OOPSLA '06, ACM, New York, NY, USA (2006), http://doi.acm.org/10.1145/1176617.1176755

[29] Tobin-Hochstadt, S., Felleisen, M.: The design and implementation of typed scheme. In: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 395–406. POPL '08, ACM, New York, NY, USA (2008), http://doi.acm.org/10.1145/1328438.1328486

[30] Torlak, E., Bodik, R.: Growing solver-aided languages with rosette. In: Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software. Onward! (2013)

[31] Vazou, N., Seidel, E.L., Jhala, R., Vytiniotis, D., Peyton-Jones, S.: Refinement types for haskell (2014)

[32] Vekris, P., Cosman, B., Jhala, R.: Refinement types for typescript (2016)

[33] Vytiniotis, D., Peyton Jones, S., Claessen, K., Rosén, D.: Halo: Haskell to logic through denotational semantics. In: Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL (2013)

[34] Weitz, K., Woos, D., Torlak, E., Ernst, M.D., Krishnamurthy, A., Tatlock, Z.: Scalable verification of border gateway protocol configurations with an smt solver. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. pp. 765–780. OOPSLA 2016, ACM, New York, NY, USA (2016), http://doi.acm.org/10.1145/2983990.2984012

[35] Xi, H., Pfenning, F.: Eliminating array bound checking through dependent types. In: Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation. PLDI (1998)