Refinement Types for Ruby

Milod Kazerounian[1], Niki Vazou[1], Austin Bourgerie[1], Jeffrey S. Foster[1], and Emina Torlak[2]

[1] University of Maryland, College Park, USA
{milod, nvazou, abourg, jfoster}@cs.umd.edu,
[2] University of Washington, Seattle, USA
emina@cs.washington.edu

Abstract. Refinement types are a popular way to specify and reason about key program properties. In this paper, we introduce RTR, a new system that adds refinement types to Ruby. RTR is built on top of RDL, a Ruby type checker that provides basic type information for the verification process. RTR works by encoding its verification problems into Rosette, a solver-aided host language. RTR handles mixins through assume-guarantee reasoning and uses just-in-time verification for metaprogramming. We formalize RTR by showing a translation from a core, Ruby-like language with refinement types into Rosette. We apply RTR to check a range of functional correctness properties on six Ruby programs. We find that RTR can successfully verify key methods in these programs, taking only a few minutes to perform verification.

Keywords: Ruby, Rosette, refinement types, dynamic languages

1 Introduction

Refinement types combine types with logical predicates to encode program invariants [32, 43]. For example, the following refinement type specification:

type : incr_sec , ‘(**Integer** x { 0 ≤ x < 60 }) → **Integer** r { 0 ≤ r < 60}’

describes a method incr_sec that increments a second. With this specification, incr_sec can only be called with integers that are valid seconds (between 0 and 59) and the method will always return valid seconds.

Refinement types were introduced to reason about simple invariants, like safe array indexing [43], but since then they have been successfully used to verify sophisticated properties including termination [39], program equivalence [9], and correctness of cryptographic protocols [28], in various languages (*e.g.,* ML [18], Racket [21], and TypeScript [40]).

In this paper, we explore refinement types for Ruby, a popular, object-oriented, dynamic scripting language. Our starting place is RDL [17, 30], a Ruby type system recently developed by one of the authors and his collaborators. We introduce RTR, a tool that adds refinement types to RDL and verifies them via a translation into Rosette [38], a solver-aided host language. Since Rosette is not object-oriented, RTR encodes Ruby objects as Rosette structs that store object fields and an integer identifying the object's class. At method calls, RTR uses RDL's type information to statically overestimate the possible callees. When methods with refinement types are called, RTR can either translate the callee directly or treat it modularly by asserting the method preconditions and assuming the postcondition, using purity annotations to determine which fields (if any) the method may mutate. (§ 2)

In addition to standard object-oriented features, Ruby includes dynamic language features that increase flexibility and expressiveness. In practice, this introduces two key challenges in refinement type verification: *mixins*, which are Ruby code modules that extend other classes without direct inheritance, and *metaprogramming*, in which code is generated on-the-fly during runtime and used later during execution. The latter feature is particularly common in Ruby on Rails, a popular Ruby web development framework.

To meet these challenges, RTR uses two key ideas. First, RTR incorporates *assume-guarantee checking* [20] to reason about mixins. RTR verifies definitions of methods in mixins by assuming refinement type specifications for all undefined, external methods. Then, by dynamically intercepting the call that includes a mixin in a class, RTR verifies the appropriate class methods satisfy the assumed refinement types (§ 3.1). Second, RTR uses *just-in-time verification* to reason about metaprogramming, following RDL's just-in-time type checking [30]. In this approach, (refinement) types are maintained at run-time, and methods are checked against their types after metaprogramming code has executed but before the methods have been called (§ 3.2).

We formalized RTR by showing how to translate λ^{RB}, a core Ruby-like language with refinement types, into λ^{I}, a core verification-oriented language. We then discuss how to map the latter into Rosette, which simply requires encoding λ^{I}'s primitive object construct into Rosette structs and translating some control-flow constructs such as `return` (§ 4).

We evaluated RTR by using it to check a range of functional correctness properties on six Ruby and Rails applications. In total, we verified 31 methods, comprising 271 lines of Ruby, by encoding them as 1,061 lines of Rosette. We needed 73 type annotations. Verification took a total median time (over multiple trials) of 506 seconds (§ 5).

Thus, we believe RTR is a promising first step toward verification for Ruby.

2 Overview

We start with an overview of RTR, which extends the Ruby type checker RDL [30] 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 to Rosette structs (§ 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, we can use RDL's **type** method to link a method with its specification:

```
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
```

This type indicates the argument and result of incr_sec are integers in the range from 0 to 59. In general, refinements (in curly braces) may be arbitrary Ruby expressions that are treated as booleans, and they should be *pure*, *i.e.,* have no side effects, since effectful predicates make verification either complicated or imprecise [41]. As in RDL, the type annotation, which is a string, is parsed and stored in a global table which maintains the program's type environment.

2.2 Verification using Rosette

RTR checks method specifications by encoding their verification into Rosette [38], a solver-aided host language built on top of Racket. Among other features, Rosette can perform verification by using symbolic execution to generate logical constraints, which are discharged using Z3 [24].

For example, to check incr_sec , RTR creates the equivalent Rosette program:

```
(define ( incr_sec x) ( if (= x 59) 0 (+ x 1)))
(define−symbolic x_in integer?)
( verify #:assume (assert 0 ≤x < 60)
         #:guarantee (assert (let ([r ( incr_sec x)]) 0 ≤ r < 60)))
```

Here x_in is an integer *symbolic constant* representing an unknown, arbitrary integer argument. Rosette symbolic constants can range over the *solvable types* integers, booleans, bitvectors, reals, and uninterpreted functions. We use Rosette's **verify** function with assumptions and assertions to encode pre- and postconditions, respectively. When this program is run, Rosette searches for an x_in such that the assertion fails. If no such value exists, then the assertion is verified.

2.3 Encoding and Reasoning about Objects

We encode Ruby objects in Rosette using a *struct type*, i.e., a record. More specifically, we create a struct type **object** that 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 (similarly to prior work [19, 34]).

For example, consider a Ruby class **Time** with three instance variables @sec, @min, and @hour, and a method is_valid that checks all three variables are valid:

```
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
```

RTR observes three fields in this program, and thus it defines:

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

Here **object** includes fields for the class ID, object ID, and the three instance variables. Note since **object**'s fields are statically defined, our encoding cannot handle dynamically generated instance variables, which we leave as future work.

RTR then translates Ruby field reads or writes as getting or setting, respectively, **object**'s fields in Rosette. For example, suppose we add a method mix to the **Time** class and specify it is only called with and returns valid times:

```
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
```

Initially, type checking fails because the getters' and setters' (*e.g.,* sec and sec=) types are unknown. Thus, we add those types:

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

(Note these annotations can be generated automatically using our approach to metaprogramming, described in § 3.2.) This allows RTR to proceed to the translation stage, which generates the following 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 )
```

(Asserts, assumes, and verify call omitted.) Here (**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** turns into an explicit additional argument in the Rosette definition. Rosette then verifies this program, thus verifying the original Ruby mix method.

2.4 Method Calls

To translate a Ruby method call e.m(e1, .., en), RTR needs to know the callee, which depends on the runtime type of the receiver e. RTR uses RDL's type information to overapproximate the set of possible receivers. For example, if e has type A in RDL, then RTR translates the above as a call to A.m. If e has a union type, RTR emits Rosette code that branches on the potential types of the receiver using **object** class IDs and dispatches to the appropriate method in each branch. This is similar to class hierarchy analysis [16], which also uses types to determine the set of possible method receivers and construct a call graph.

Once the method being called is determined, we translate the call into Rosette. As an example, consider a 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
```

RTR's translation of to_sec could simply call directly into incr_sec's translation. This is equivalent to inlining incr_sec's code. However, inlining is not always possible or desirable. A method's code may not be available because the method comes from a library, is external to the environment (§ 3.1), or has not been defined yet (§ 3.2). The method might also contain constructs that are difficult to verify, like diverging loops.

Instead, RTR can model the method call using the programmer provided method specification. To precisely reason with only a method's specification, RTR follows Dafny [22] and treats pure and impure methods differently.

Pure methods. Pure methods have no side effects and return the same result for the same inputs, satisfying the congruence property $\forall x, y. x = y \Rightarrow m(x) = m(y)$ for a given method m. Thus, pure methods can be encoded using Rosette's uninterpreted functions. The method incr_sec is indeed pure, so we can label it as such:

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

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

Impure methods. Most Ruby methods have side effects and thus are not pure. For example, consider incr_min, a mutator method that adds a minute to a **Time**:

```
type ‘(Time t { t. is_valid ∧ t.min < 59 }) → Time r { r. is_valid }’,
  modifies: { t: @min, t: @sec }
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
```

A translated call to incr_min generates a fresh symbolic value as the method's output and assumes the method's postcondition on that value. Because the method may have side effects, the **modifies** label is used to list all fields of inputs which may be modified by the method. Here, a translated call to incr_min will *havoc* (set to fresh symbolic values) t's @min and @sec fields.

We leave support for other kinds of modifications (e.g., global variables, transitively reachable fields), as well as enforcing the **pure** and **modifies** labels, as future work.

3 Just-In-Time Verification

Next, we show how RTR handles code with dynamic bindings via 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 that are added to any class that **include**s the module at runtime. Interestingly, modules may refer to methods that are not defined in the module but will ultimately be defined in the including class. Such incomplete environments pose a challenge for verification.

Consider the following method that has been extracted and simplified from the Money library described in § 5.

```
module Arithmetic
  type '(Integer x)→ Float r { r==x/value }'
  def div_by_val (x) x/value; end
end
```

The module method div_by_val divides its input x by value. RTR's specification for / requires that value cannot be 0.

Notice that value is not defined in **Arithmetic**. Rather, it must be defined wherever **Arithmetic** is included. Therefore, to proceed with verification in RTR, the programmer must provide an annotation for value:

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

Using this annotation, RTR can verify div_by_value . Then when **Arithmetic** is included in another class, RTR verifies value's refinement type. For example, consider the following code:

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

RTR dynamically intercepts the call to **include** and then applies the type annotations for methods not defined in the included module, in this case verifying value against the annotation in **Arithmetic**. Thus, RTR follows an assume-guarantee paradigm [20]: it assumes value's annotation while verifying div_by_val and then guarantees the annotation once value is defined.

3.2 Metaprogramming

Metaprogramming in Ruby allows new methods to be created and existing methods to be redefined on the fly, posing a challenge for verification. RTR addresses this challenge using just-in-time checking [30], in which, in addition to code, method annotations can also be generated dynamically.

We illustrate the just-in-time approach using an example from Boxroom, a Rails app for managing and sharing files in a web browser (§ 5). The app defines a class **UserFile** that is a Rails *model* corresponding to a database table:

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

Here calling belongs_to tells Rails that every **UserFile** is associated with a folder (another model). The move method updates the associated folder of a **UserFile** and saves the result to the database. We annotate move to specify that the **UserFile**'s new folder should be the same as move's argument.

This method and its annotation are seemingly simple, but there is a problem. To verify move, RTR needs an annotation for the folder= method, which is not statically defined. Rather, it is dynamically generated by the call to belongs_to.

To solve this problem in RTR, we instrument belongs_to to generate type annotations for the setter (and getter) method, as follows:

```
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 }’
    true
  end
end
```

We call **pre**, an RDL method, to define a precondition *code block* (*i.e.,* an anonymous function) which will be executed on each call to belongs_to. First, the block sets name and cname to be the string version of the first argument passed to belongs_to and its camelized representation, respectively. Then, we create types for the name and name= methods. Finally, we return true so the contract will succeed. In our example, this code generates the following two type annotations upon the call to belongs_to:

```
type ‘folder’, ‘() → Folder c’, :pure
type ‘folder=’, ‘(Folder i) → Folder o { folder == i }’
```

With these annotations, verification of the initial move method succeeds.

4 From Ruby to Rosette

In this section, we formally describe our verifier and the translation from Ruby to Rosette. We start (§ 4.1) by defining λ^{RB}, a Ruby subset that is extended with refinement type specifications. We give (§ 4.2) a translation from λ^{RB} to an intermediate language λ^{I}, and 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.

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

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

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

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

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

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

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

λ^{RB}. Figure 1 defines λ^{RB}, a core Ruby-like language with refinement types. *Constants* consist of `nil`, booleans, and integers. *Expressions* include constants, variables, assignment, conditionals, sequences, and the reserved variable `self`, which refers to a method's receiver. Also included are references to an instance variable f and instance variable assignment; we note that in actual Ruby, field names are preceded by a "@". Finally, expressions include method calls, constructor calls A.`new` which create a new instance of class A, and return statements.

Refined types $\{x : A \mid e\ \}$ refine the basic type A with the predicate e. The basic type A is used to represent both user defined and built-in classes including nil, booleans, integers, floats, etc. The refinement e is a *pure, boolean valued* expression that may refer to the refinement variable x. In the interest of greater simplicity for the translation, we require that `self` *does 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 `def` $A.m(\{x_1 : A_1 \mid e_1\ \}, \ldots, \{x_n : A_n \mid e_n\ \})::t; l =$

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 arguments' refinements types e_j for all $1 \leq j \leq n$, and can also appear in the refinement of the result type t. A method type annotation $A.m :: (\overline{t}) \rightarrow t\ ;\ l$ binds the method named $A.m$ with the dependent function type $(\overline{t}) \rightarrow t$. λ^{RB} includes both method annotations and method definitions because annotations are used when a method's code is not available, *e.g.,* in the cases of library methods, mixins, or metaprogramming.

A *label* l can appear in both method definitions and annotations to direct the method's translation into Rosette as described in § 2.4. 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 `modifies`$[\overline{x.f}]$ label is used when a method is impure, *i.e.,* it may modify its inputs. As we saw earlier, the list $\overline{x.f}$ captures all the argument fields which the method may modify.

$\boldsymbol{\lambda^I}$. Figure 2 defines λ^I, a core verification-oriented language that easily translates to Rosette. λ^{RB} methods map to λ^I functions, and λ^{RB} objects map to a special `object` struct type. λ^I provides primitives for creating, altering, and referencing instances of this type. *Values* in λ^I consist of *constants* c (defined identically to λ^{RB}) and `object`$(i_1,\ i_2,\ [f_1\ w_1] \ldots [f_n\ w_n])$, an instantiation of an `object` type with class ID i_1, object ID i_2, and where each field f_i of the `object` is bound to value w_i. *Expressions* include `let` bindings (`let` $(\overline{[x_i\ u_i]})$ `in` u) where each x_i may appear free in u_j if $i < j$. They also include function calls, `assert`, `assume`, and `return` statements, as well as `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 $x.f$.

A *program* is a series of definitions and verification queries. A *definition* is a function definition or a symbolic definition `define-sym`$(x,\ A)$, which binds x to either a fresh symbolic value if A is a solvable type (e.g., boolean, integer; see § 2.2) or a new `object` with symbolic fields defined depending on the type of A. Finally, a verification query `verify`$(\overline{u} \Rightarrow u)$ checks the validity of u assuming $\overline{u}$.

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:

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

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

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

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

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

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

$$\frac{\texttt{classId}(A) = i_c \quad \texttt{freshID}(i_o) \quad f_i \in \mathcal{F}}{A.\texttt{new} \rightsquigarrow \texttt{object}(i_c,\ i_o,\ [f_1\ \texttt{nil}]\ldots[f_{|\mathcal{F}|}\ \texttt{nil}])}\ \textsc{T-New}$$

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

$$\frac{\begin{array}{c}\texttt{typeOf}(e_F) = A \qquad \texttt{pure} = \texttt{labelOf}(A.m)\\ A\_m \in \mathcal{U} \qquad \texttt{freshVar}(x, r)\\ \texttt{specOf}(A.m) = (\{x : A_x \mid e_x\ \}) \to \{r : A_r \mid e_r\ \}\\ e_F \rightsquigarrow u_F \quad e \rightsquigarrow u \quad e_x \rightsquigarrow u_x \quad e_r \rightsquigarrow u_r\end{array}}{e_F.m(e) \rightsquigarrow \texttt{let}\ ([x\ u][r\ A\_m(u_F, a)])\ \texttt{in}\ \texttt{assert}(u_x)\ ;\ \texttt{assume}(u_r)\ ;\ r}\ \textsc{T-Pure1}$$

$$\frac{\begin{array}{c}\texttt{typeOf}(e_F) = A \qquad \texttt{modifies}[p] = \texttt{labelOf}(A.m)\\ \texttt{specOf}(A.m) = (\{x : A_x \mid e_x\ \}) \to \{r : A_r \mid e_r\ \}\\ h_x = \{u.f \mid f \in \mathcal{F},\ x.f \in p\} \quad h_F = \{u_F.f \mid f \in \mathcal{F},\ \texttt{self}.f \in p\}\\ \texttt{freshVar}(x, r) \quad e_F \rightsquigarrow u_F \quad e \rightsquigarrow u \quad e_x \rightsquigarrow u_x \quad e_r \rightsquigarrow u_r\end{array}}{e_F.m(e) \rightsquigarrow \begin{array}{l}\texttt{let}\ ([x\ u])\ \texttt{in}\ \texttt{define-sym}(r,\ A_r);\\ \texttt{assert}(u_x)\ ;\ \texttt{havoc}(h_F \cup h_x)\ ;\ \texttt{assume}(u_r)\ ;\ r\end{array}}\ \textsc{T-Impure1}$$

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

$$\frac{}{\cdot \rightsquigarrow \cdot}\ \textsc{T-Emp} \qquad \frac{P \rightsquigarrow Q}{A.m :: (x_1{:}t_1, \ldots, x_n{:}t_n) \to t\ ;\ l, P \rightsquigarrow Q}\ \textsc{T-Ann}$$

$$\frac{\begin{array}{c}t_i = \{x_i : A_{x_i} \mid e_{x_i}\ \} \qquad t = \{r : A_r \mid e_r\ \}\\ e \rightsquigarrow u \quad e_{x_i} \rightsquigarrow u_{x_i} \quad e_r \rightsquigarrow u_r \quad P \rightsquigarrow Q \quad 1 \le i \le n\end{array}}{\texttt{def}\ A.m(t_1, \ldots, t_n){::}t; l = e, P \rightsquigarrow \begin{array}{l}\texttt{define}\ A\_m(\mathit{self}, x_1, \ldots, x_n) = u;\\ \texttt{define-sym}(self,\ A);\\ \texttt{define-sym}(x_i,\ A_{x_i});\\ \texttt{verify}(u_{x_1}, \ldots, u_{x_n} \Rightarrow u_r)\ ;\ Q\end{array}}\ \textsc{T-Def}$$

Fig. 3. Translation from λ^{RB} to λ^{I}. For simplicity rules T-Pure1 and T-Impure1 assume single argument methods.

In the translation rules, we use the standard set operations $x \in \mathcal{X}$ and $\mid \mathcal{X} \mid$ to check membership and size of the set $\mathcal{X}$. Thus, the translation relation is defined over these sets: $\mathcal{M}, \mathcal{U}, \mathcal{F} \vdash e \rightsquigarrow u$. Since the rules do not modify these environments, in Figure 3 we simplify the rules to $e \rightsquigarrow u$. Note that even though the rules "guess" these environments by making assumptions about which elements are members of the sets, in an algorithmic definition the rules can be used 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 read from and write to, respectively, 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 premise 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`, $f_1, \dots, f_{|\mathcal{F}|}$, is initially bound to `nil`.

Method Calls. To translate the λ^{RB} method call $e_F.m(\overline{e})$, we first use the function `typeOf`(e_F) to type e_F via RDL type checking [30]. 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 specified in the annotation of $A.m$ (as informally described in § 2.4).

The rule T-Exact is used when the label is `exact`. 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. Moreover, $A.m$ 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 a natural way to account for more arguments. The rule T-Pure1 is used when the label is `pure`. In this case, the call is translated as an invocation to the uninterpreted function A_m, so $A.m$ should be in the global set of uninterpreted functions $\mathcal{U}$. The specification `specOf`$(A.m)$ of the method is also enforced. Let $(\{x : A_x \mid e_x\}) \rightarrow \{r : A_r \mid e_r\}$ be the specification. We assume that the binders in the specification are α-renamed so that the binders x and r are fresh. We use x and r to bind the argument and the result, respectively, to ensure, via A-normal form conversion [33], that they will be evaluated exactly once, even though x and r may appear many times in the refinements. To enforce the specification, we assert the method's precondition e_x and assume the postcondition e_r.

If a method is labeled with `modifies`$[p]$ then the rule T-Impure1 is applied. We locally define a new symbolic object as the return value, and we `havoc` the fields of all arguments (including *self*) specified in the `modifies` 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 discards 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 value 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 $\boldsymbol{\lambda^{I}}$ to Rosette

We write $Q \twoheadrightarrow R$ to encode the translation of the λ^{I} program Q to the Rosette program R. This translation is straightforward, since λ^{I} consists of Rosette extended with some macros to encode Ruby-verification specific operators, like `define-sym` 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 are expanded in Rosette to perform the associated operations over `object` structs. The expressions `object`$(i_c,\ i_o,\ \overline{[f\ w]})$ and `havoc`$(x.f)$, and the definition `define-sym`$(x,\ A)$, all described in § 4.1, are expanded to perform the corresponding operations over values of the `object` struct type.

Control Flow. Macro expansion is used to translate `return` and `assume` statements, and exceptions into Rosette, since those forms are not built-in to the language. To encode `return`, we expand every function definition in λ^{I} to keep track of a local variable `ret`, which is initialized to a special `undefined` value and returned at the end of the function. We transform every statement `return`(e) to update the value of `ret` to e. Then, we expand every expression u in a function to `unless-done`(u), which checks the value of `ret`, proceeding with u if `ret` is `undefined` or skipping u if there is a return value.

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. A similar expansion is used to encode and propagate exceptions.

4.4 Primitive Types

λ^{RB} provides constructs for functions, assignments, control flow, etc, but does not provide the theories 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 encode certain Ruby expressions, such as constants c in λ^{RB}, into Rosette's corresponding built-in datatypes.

Equality and Booleans. To precisely reason about equality, we encode Ruby's `==` method over arbitrary objects using the object class' == method if one is defined. If the class inherits this method from Ruby's top class, *BasicObject*, then we encode == using Rosette's equality operator `equal?` to check equality of object IDs. We encode Ruby's booleans and operations over them as Rosette's respective booleans and their operators.

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. To perform (bounded) verification in such cases, we provide, via a configuration flag, the option of encoding Ruby's integers as Rosette's built-in finite sized bitvectors.

Arrays. Finally, we provide a special encoding for Ruby's arrays, which are commonly used both for storing arbitrarily large random-access data and to represent mixed-type tuples, stacks, queues, etc. We encode Ruby's arrays as a Rosette struct composed of a fixed-size vector and an integer representing the current size of the Ruby array. Because we used fixed-size vectors, we can only perform bounded verification over arrays. On the other hand, we avoid the need for loop invariants for iterators and reasoning over array operations can be more efficient.

4.5 Verification of λ^{RB}

We define a verification algorithm RTR$^{\lambda}$ that, given a λ^{RB} program P, checks if all the definitions satisfy their specifications. The pseudo-code for this algorithm is shown below:

```
def RTR$^{\lambda}$($P$)
  ($\mathcal{F}$, $\mathcal{U}$, $\mathcal{M}$) := guess($P$)
  for ($f \in \mathcal{F}$): add field $f$ to object struct
  for ($u \in \mathcal{U}$): define uninterpreted function $u$
  $P \rightsquigarrow Q \twoheadrightarrow R$
  return if (valid(R)) then SAFE else UNSAFE
end
```

First, we `guess` the proper translation environments. In practice (as discussed in § 4.2), we use the translation of P to generate the minimum environments for which translation of P succeeds. We define an `object` struct in Rosette containing one field for each member of $\mathcal{F}$, and we define an uninterpreted function for each method in $\mathcal{U}$. Next, we translate P to a λ^{I} program Q via $P \rightsquigarrow Q$ (§ 4.2) and Q to a the Rosette program R, via $Q \twoheadrightarrow R$ (§ 4.3). Finally, we run the Rosette program R. The initial program P is *safe*, *i.e.,* no refinement type specifications are violated, if and only if the Rosette program R is *valid*, *i.e.,* all the `verify` queries are valid.

We conclude this section with a discussion of the RTR$^{\lambda}$ verifier.

RTR$^{\lambda}$ is Partial. There exist expressions of λ^{RB} that fail to translate into a λ^{I} expression. The translation requires at each method call $e_F.m(\overline{e})$ that the receiver has a class type A. There are two cases where this requirement fails: (1) e_F has a union type or (2) type checking fails and so e_F has no type. In our implementation (§ 5), we extend the translation to handle the first two cases. Handling for (1) is outlined in § 2.4. Case (2) can be caused by either a type error in the program or a lack of typing information for the type checker. Translation cannot proceed in either case.

RTR$^{\lambda}$ may Diverge. The translation to Rosette always terminates. All translation rules are inductively defined: they only recurse on syntactically smaller expressions or programs. Also, since the input program is finite, the minimum global environments required for translation are also finite. Finally, all the helper functions (including the type checking `typeOf`$(\cdot)$) do terminate.

Yet, verification may diverge, as the execution of the Rosette program may diverge. Specifications can encode arbitrary expressions, thus it is possible to encode undecidable verification queries. Consider, for instance, the following contrived 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 attempts to symbolically evaluate this program, and thus diverges.

RTR$^{\lambda}$ 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 axiom.

RTR$^{\lambda}$ is Sound. If the verifier decides that the input program is safe, then all definitions satisfy their specifications, assuming that (1) all the refinements are pure boolean expressions and (2) all the labels are sound (i.e., methods match the specifications implied by the labels). The assumption (1) is required since verification under diverging (let alone effectful) specifications is difficult [41]. The assumption (2) is required since our translation encodes pure methods as uninterpreted functions, while for the impure methods it havocs only the unprotected arguments.

5 Evaluation

We implemented the Ruby refinement type checker RTR[3] by extending RDL [30] with refinement types. Table 1 summarizes the evaluation of RTR.

[3] Code available at: `https://github.com/mckaz/vmcai-rdl`

Benchmarks. We evaluate RTR on six popular Ruby libraries:

– `Money` [6] performs currency conversions over monetary quantities and relies on mixin methods,
– `BusinessTime` [3] performs time calculations in business hours and days,
– `Unitwise` [7] performs various unit conversions,
– `Geokit` [4] performs calculations over locations on Earth,
– `Boxroom` [2] is a Rails app for sharing files in a web browser and uses metaprogramming, and
– `Matrix` [5] is a Ruby standard library for matrix operations.

For verification, we forked the original Ruby libraries and provided manually written method specifications in the form of refinement types. The forked repositories are publicly available [8]. Experiments were conducted on a machine with a 3 GHz Intel Core i7 processor and 16 GB of memory.

We chose these libraries because they combine Ruby-specific features challenging for verification, like metaprogramming and mixins, with arithmetic-heavy operations. In all libraries we verify both (1) *functional correctness of arithmetic operations* (*e.g.,* no division-by-zero, the absolute value of a number should not be negative) and (2) *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). In the `Matrix` library, we verify a matrix multiplication method, checking that multiplying a matrix with r rows by a matrix with c columns yields a matrix of size $r \times c$. Note this method makes extensive use of array operations, since matrices are implemented as an array of arrays.

Quantitative Evaluation. Table 1 summarizes our evaluation quantitatively. For each application, we list every verified **Method**. In our experiment, we focused on methods with interesting arithmetic properties.

The **Ruby LoC** column gives the size of the verified Ruby program. This metric includes the lines of all methods and annotations that were used to verify the method in question. For each verified method, RTR generates a separate Rosette program. We give the sizes of these resulting programs in the **Rosette LoC** column. Unsurprisingly, the LoC of the Rosette 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 verified method, the **SIQR** was at most 2% of the verification time, indicating relatively little variance in the verification time. Overall, verification was fast, as might be expected for relatively small methods. The one exception was matrix multiplication. In this case, the slowdown was due to the extensive use of array operations mentioned above. We bounded array size (see § 4.4) at 10 for the evaluations. For symbolic arrays, this means Rosette must reason about every possible size of an array up to 10. This burden is exacerbated by matrix multiplication's use of two symbolic two-dimensional arrays.

Table 1. Method gives the class and name of the method verified. **Ru-LoC** and **Ro-LoC** give number of LoC for a Ruby method and the translated Rosette program. **Spec** is the number of method and variable type annotations we had to write. **Verification Time** is the median and semi-interquartile range of the time in seconds over 11 runs. **App Total** rows list the totals for an app, without double counting the same specs.

Method	**Ru-LoC**	**Ro-LoC**	**Spec**	**Verification Time**	
				Time(s)	**SIQR**
Money					
Money::Arithmetic#-@	7	29	4	5.69	0.14
Money::Arithmetic#eql?	11	40	3	5.74	0.03
Money::Arithmetic#positive?	5	24	3	5.40	0.01
Money::Arithmetic#negative?	5	24	2	5.42	0.01
Money::Arithmetic#abs	5	30	4	5.49	0.01
Money::Arithmetic#zero?	5	26	2	5.38	0.02
Money::Arithmetic#nonzero?	5	24	2	5.43	0.03
App Total	**43**	**197**	**10**	**38.56**	**0.25**
BusinessTime					
ParsedTime#-	10	58	8	6.28	0.02
BusinessHours#initialize	5	26	2	5.36	0.04
BusinessHours#non_negative_hours?	5	26	2	5.4	0.01
Date#week	7	32	2	5.53	0.01
Date#quarter	5	28	2	5.47	0.00
Date#fiscal_month_offset	5	25	2	5.41	0.02
Date#fiscal_year_week	7	33	2	5.53	0.03
Date#fiscal_year_month	12	35	3	5.65	0.02
Date#fiscal_year_quarter	9	42	2	5.72	0.03
Date#fiscal_year	11	32	4	5.81	0.03
App Total	**76**	**337**	**24**	**56.15**	**0.20**
Unitwise					
Unitwise::Functional.to_cel	4	25	2	5.42	0.03
Unitwise::Functional.from_cel	4	25	2	5.44	0.03
Unitwise::Functional.to_degf	4	22	1	5.41	0.01
Unitwise::Functional.from_degf	4	27	2	5.44	0.02
Unitwise::Functional.to_degre	4	27	2	5.44	0.01
Unitwise::Functional.from_degre	4	27	2	5.42	0.01
App Total	**24**	**153**	**6**	**32.55**	**0.11**
Geokit					
Geokit::Bounds#center	7	31	4	5.4	0.02
Geokit::Bounds#crosses_meridian?	7	35	6	5.59	0.12
Geokit::Bounds#==	9	60	5	5.97	0.13
Geokit::GeoLoc#province	5	26	2	5.52	0.11
Geokit::GeoLoc#success?	5	26	2	5.51	0.05
Geokit::Polygon#contains?	26	68	10	10.8	0.07
App Total	**59**	**246**	**21**	**38.80**	**0.50**
Boxroom					
UserFile#move	12	34	3	5.57	0.05
Matrix					
Matrix.*	57	94	9	334.35	3.99
Total	**271**	**1061**	**73**	**505.98**	**5.10**

Finally, Table 1 lists the number of type **specifications** required to verify each method. These are comprised of method type annotations, including the refinement type annotations for the verified methods themselves, and variable type annotations for instance variables. Note that we do not quantify the number of type annotations used for Ruby's core and standard libraries, since these are included in RDL.

We observe that there is significant variation in the number of annotations required for each application. For example, `Unitwise` required 6 annotations to verify 6 methods, while `Geokit` required 21 annotations for 6 methods. The differences are due to code variations: To verify a method, the programmer needs to give a refinement type for the method plus a type for each instance variable used by the method and for each (non-standard/core library) method called by the method.

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 example, 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)
  if 0 == @count
    @min = data ; @max = data
  else
    @max = data if data > @max ; @min = data if data < @min
  end
  @count += 1 ; @sum += data ; @sum2 += (data * data)
  @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
```

Here, the symbol :bind is an arbitrary label. To verify the specification, we load the library and call the verifier with this label:

```
rdl_do_verify :bind
```

RTR proceeds with verification in three steps:

- first use RDL to type check 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.

Initially, verification fails in the first step 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 following type annotations.

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

The << method also calls two methods that 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 if the given data is an outlier based on provided specifications from the programmer. These methods are challenging to verify. For example, the `to_index` method makes use of nonlinear arithmetic in the form of logarithms, and it includes a loop. 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 postcondition on <<, and it will return the following 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 that 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 on Rails. Several prior systems can verify properties of Rails apps. *Space* [26] detects security bugs in Rails apps by using symbolic execution to generate a model of data exposures in the app and reporting a bug if the model does not match common access control patterns. Bocić and Bultan proposes *symbolic model extraction* [14], which extracts models from Rails apps at runtime, to handle metaprogramming. The generated models are then used to verify data integrity and access control properties. *Rubicon* [25] allows programmers to write specifications using a domain-specific language that looks similar to Rails tests, but with the ability to quantify over objects, and then checks such specifications with bounded verification. *Rubyx* [15] likewise allows programmers to write their own specifications over Rails apps and uses symbolic execution to verify these specifications.

In contrast to RTR, all of these tools are specific to Rails and do not apply to general Ruby programs, and the first two systems do not allow programmers to specify their own properties to be verified.

Rosette. Rosette has been used to help establish the security and reliability of several real-world software systems. Pernsteiner et al. [27] use Rosette to build a verifier to study the safety of the software on a radiotherapy machine. *Bagpipe* [42] builds a verifier using Rosette to analyze the routing protocols used by Internet Service Providers (ISPs). These results show that Rosette can be applied in a variety of domains.

Types For Dynamic Languages. There have been a number of efforts to bring type systems to dynamic languages including Python [10, 12], Racket [36, 37], and JavaScript [11, 23, 35], among others. However, these systems do not support refinement types.

Some systems have been developed to introduce refinement types to scripting and dynamic languages. *Refined TypeScript* (RSC) [40] introduces refinement types to TypeScript [13, 29], a superset of JavaScript that includes optional static typing. RSC uses the framework of Liquid Types [31] to achieve refinement inference. Refinement types have been introduced [21] to Typed Racket as well. As far as we are aware, these systems do not support mixins or metaprogramming.

General Purpose Verification Dafny [22] 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 RTR largely emulates. However, unlike Dafny, RTR leaves this modular treatment of methods as an option for the programmer. Furthermore, unlike RTR, Dafny does not include features such as mixins and metaprogramming.

7 Conclusion and Future Work

We formalized and implemented RTR, a refinement type checker for Ruby programs using assume-guarantee reasoning and 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 numeric properties on six commonly used Ruby and Ruby on Rails applications, by adding refinement type specifications to the existing method definitions. We found that verifying these methods took a reasonable runtime and annotation burden, and thus we believe RTR is a promising first step towards bringing verification to Ruby.

Our work opens new directions for further Ruby verification. We plan to explore verification of purity and immutability labels, which are currently trusted by RTR. We also plan to develop refinement type inference by adapting Hindley-Milner and liquid typing [31] to the RDL setting, and by exploring whether Rosette's synthesis constructs could be used for refinement inference. We will also extend the expressiveness of RTR by adding support for loop invariants and dynamically defined instance variables, among other Ruby constructs. Finally, as Ruby is commonly used in the Ruby on Rails framework, we will extend RTR with modeling for web-specific constructs such as access control protocols and database operations to further support verification in the domain of web applications.

Acknowledgements

We thank Thomas Gilray and the anonymous reviewers for their feedback on earlier versions of this paper. This work is supported in part by NSF CCF-1319666, CCF-1518844, CCF-1618756, CCF-1651225, CNS-1518765, and DGE-1322106.

Bibliography

[1] Aggregate (2017), https://github.com/josephruscio/aggregate

[2] Boxroom (2017), https://github.com/mischa78/boxroom

[3] Businesstime (2017), https://github.com/bokmann/business_time/

[4] Geokit (2017), https://github.com/geokit/geokit

[5] Matrix (2017), https://github.com/ruby/matrix

[6] Money (2017), https://github.com/RubyMoney/money

[7] Unitwise (2017), https://github.com/joshwlewis/unitwise/

[8] Verified ruby apps (2017), https://raw.githubusercontent.com/mckaz/milod.kazerounian.github.io/master/static/VMCAI18/source.md

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

[10] Ancona, D., Ancona, M., Cuni, A., Matsakis, N.D.: Rpython: A step towards reconciling dynamically and statically typed oo languages. DLS (2007)

[11] Anderson, C., Giannini, P., Drossopoulou, S.: Towards type inference for javascript. ECOOP (2005)

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

[13] Bierman, G., Abadi, M., Torgersen, M.: Understanding typescript. ECOOP (2014)

[14] Bocić, I., Bultan, T.: Symbolic model extraction for web application verification. ICSE (2017)

[15] Chaudhuri, A., Foster, J.S.: Symbolic security analysis of ruby-on-rails web applications. CCS (2010)

[16] Dean, J., Grove, D., Chambers, C.: Optimization of object-oriented programs using static class hierarchy analysis. ECOOP (1995)

[17] Foster, J., Ren, B., Strickland, S., Yu, A., Kazerounian, M.: RDL: Types, type checking, and contracts for Ruby (2017), https://github.com/plum-umd/rdl

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

[19] Jeon, J., Qiu, X., Foster, J.S., Solar-Lezama, A.: Jsketch: Sketching for java. ESEC/FSE 2015 (2015)

[20] Jones, C.: Specification and design of (parallel) programs. IFIP Congress (1983)

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

[22] Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. LPAR (2010)

[23] Lerner, B.S., Politz, J.G., Guha, A., Krishnamurthi, S.: Tejas: Retrofitting type systems for javascript (2013)

[24] de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. TACAS (2008)

[25] Near, J.P., Jackson, D.: Rubicon: Bounded verification of web applications. FSE '12 (2012)

[26] Near, J.P., Jackson, D.: Finding security bugs in web applications using a catalog of access control patterns. ICSE '16 (2016)

[27] 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. CAV (2016)

[28] 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)

[29] Rastogi, A., Swamy, N., Fournet, C., Bierman, G., Vekris, P.: Safe & efficient gradual typing for typescript (2015)

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

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

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

[33] Sabry, A., Felleisen, M.: Reasoning about programs in continuation-passing style. LFP '92 (1992)

[34] Singh, R., Gulwani, S., Solar-Lezama, A.: Automated feedback generation for introductory programming assignments. PLDI (2013)

[35] Thiemann, P.: Towards a type system for analyzing javascript programs. ESOP (2005)

[36] Tobin-Hochstadt, S., Felleisen, M.: Interlanguage migration: From scripts to programs. OOPSLA (2006)

[37] Tobin-Hochstadt, S., Felleisen, M.: The design and implementation of typed scheme. POPL (2008)

[38] Torlak, E., Bodik, R.: Growing solver-aided languages with rosette. Onward! (2013)

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

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

[41] Vytiniotis, D., Peyton Jones, S., Claessen, K., Rosén, D.: Halo: Haskell to logic through denotational semantics. POPL (2013)

[42] Weitz, K., Woos, D., Torlak, E., Ernst, M.D., Krishnamurthy, A., Tatlock, Z.: Scalable verification of border gateway protocol configurations with an smt solver. OOPSLA (2016)

[43] Xi, H., Pfenning, F.: Eliminating array bound checking through dependent types. PLDI (1998)