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

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

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

Refinement Types for Ruby

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

University of Maryland [1]University of Washington

1 From Ruby to Rosette

In this section we formally describe our verifier that translates Ruby to Rosette programs. We start (§ 1.1) by defining λ^{RB} as the subset of Ruby that is important in our translation extended with refinement type specifications. We achieve the translation to Rosette by first translating λ^{RB} to an intermediate language λ^{I} (§ 1.2). Then (§ 1.3), we discuss how λ^{I} maps to a Rosette program. Finally (§ 1.5), we use this translation to construct a verifier for Ruby programs.

1.1 Core Ruby λ^{RB} & 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, instance variable assignment, 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 classes and all primitive types including nil, booleans, integers, floats, *etc.*. The refinement expression e is a *pure, boolean valued* that may include the refinement variable x of type A. 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` *does not* appear

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

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

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

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` $A.m(x_1 : t_1, \ldots, x_n : t_n) :: t; l = \ e$ 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 `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.

$\boldsymbol{\lambda^I}$ Figure 2 defines λ^I, a core verification-oriented language that easily translates (§ 1.3) 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 `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 (`let` $(\overline{[x_i\ u_i]})$ `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 `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 `define-obj`$(x, i_1,\ i_2, [x_1\ u_1] \dots [x_n\ u_n])$, 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 `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 `verify`$(\overline{u} \Rightarrow u)$ checks the validity of u assuming $\overline{u}$.

1.2 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:

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

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 [1]. 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.

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

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

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

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

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

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

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

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

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

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

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

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

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.

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 [2]. 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.

1.3 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 (§ **??**) 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.

1.4 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 and operations over them 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.

1.5 Verification of λ^{RB}

We define a verification algorithm RTR MK:♣ use different name for algorithm?♣ 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 $P \rightsquigarrow Q$ (§ 1.2). Next, it translates Q to a Rosette program via macro expansion (§ 1.3). 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
```

MK:♣ Something in the algorithm to express macro expansion?♣

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.,* `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 § **??**. 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 `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 [3] 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.

Bibliography

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

[2] 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`

[3] 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)