

PROVING TRANSLATION FAITHFULNESS  
AND TYPE SOUNDNESS OF PRUBY  
USING COQ

Zhongqiang Huang and Daehwan Kim

# A brief introduction to Ruby



- One of popular scripting languages like Python and Perl
- Supports multiple programming paradigms
  - ▣ Functional
  - ▣ Object oriented
  - ▣ Imperative
  - ▣ Reflective
- Dynamic typed
- Provides highly dynamic constructs like eval

# Why static typing analysis is difficult in Ruby ?

- Eval makes static typing analysis difficult

```
>> a = 1  
⇒ 1
```

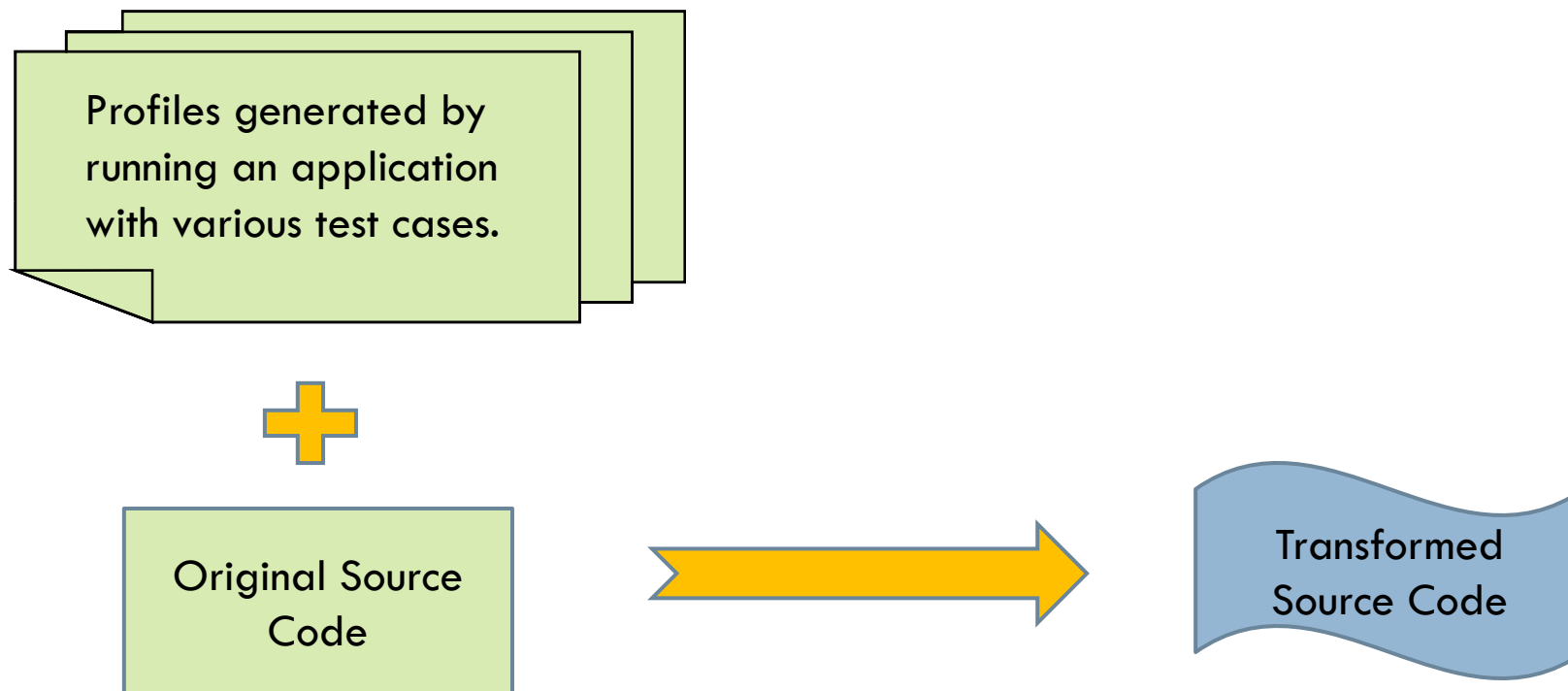
```
>> eval 'a+1'  
⇒ 2
```

```
>> $input = Some string "???" from  
networks.  
⇒ "???"
```

```
>> eval $input  
⇒ Something unexpected output!
```

# PRuby: Profile-guided static typing

- But there is a way to get around the problem
  - ▣ Most usages of such dynamic features are heavily constrained
  - ▣ Many test suites are available and testing is common practice in the Ruby community



# PRuby: Profile-guided static typing

- But there is a way to generate profiles
  - ▣ Most usages of such dynamic typing are in the Ruby community
  - ▣ Many test suites are available in the Ruby community

Profiles generated by running an application with various test cases.



Original Source Code

<p>(VAR)</p> $\frac{}{\langle M, V, x \rangle \rightarrow \langle M, \emptyset, V(x) \rangle}$	<p>(DEF)</p> $\frac{}{\langle M, V, d \rangle \rightarrow \langle (d, M), \emptyset, \text{false} \rangle}$
--	---

(EVAL)

$$\frac{\langle M, V, e \rangle \rightarrow \langle M_1, \mathcal{P}_1, s \rangle \quad \langle M_1, V, \text{parse}(s) \rangle \rightarrow \langle M_2, \mathcal{P}_2, v \rangle}{\langle M, V, \text{eval}_\ell e \rangle \rightarrow \langle M_2, (\mathcal{P}_1 \cup \mathcal{P}_2 \cup [\ell \mapsto s]), v \rangle}$$

(SEND)

$$\frac{\langle M, V, e_1 \rangle \rightarrow \langle M_1, \mathcal{P}_1, s \rangle \quad m = \text{parse}(s) \quad \langle M_1, V, e_0.m(e_2, \dots, e_n) \rangle \rightarrow \langle M_2, \mathcal{P}_2, v \rangle}{\langle M, V, e_0.\text{send}_\ell(e_1, \dots, e_n) \rangle \rightarrow \langle M_2, (\mathcal{P}_1 \cup \mathcal{P}_2 \cup [\ell \mapsto s]), v \rangle}$$

(CALL-M)

$$\frac{\langle M_i, V, e_i \rangle \rightarrow \langle M_{i+1}, \mathcal{P}_i, v_i \rangle \quad i \in 0..n \quad v_0 = \text{new } A \quad (\text{def}_\ell A.m(\dots) = \dots) \notin M_{n+1} \quad (\text{def}_{\ell'} A.\text{method\_missing}(x_1, \dots, x_{n+1}) = e) \in M_{n+1} \quad s = \text{unparse}(m) \quad m \neq \text{method\_missing} \quad V' = [\text{self} \mapsto v_0, x_1 \mapsto s, x_2 \mapsto v_1, \dots, x_{n+1} \mapsto v_n] \quad \langle M_{n+1}, V', e \rangle \rightarrow \langle M', \mathcal{P}', v \rangle}{\langle M_0, V, e_0.m(e_1, \dots, e_n) \rangle \rightarrow \langle M', (\bigcup_i \mathcal{P}_i) \cup \mathcal{P}' \cup [\ell' \mapsto s], v \rangle}$$

# PRuby: Profile

- But there is a way to
  - ▣ Most usages of such code
  - ▣ Many test suites are created by the Ruby community

Profiles generated by running an application with various test cases.



Original Source Code



Transformed Source Code

$$\frac{\text{(REFL}_{\rightsquigarrow})}{\mathcal{P} \vdash e \rightsquigarrow e} \quad e \in \{x, v, \text{blame } \ell\}$$

$$\frac{\text{(SEQ}_{\rightsquigarrow})}{\mathcal{P} \vdash e_1; e_2 \rightsquigarrow e'_1; e'_2} \quad \begin{array}{l} \mathcal{P} \vdash e_1 \rightsquigarrow e'_1 \\ \mathcal{P} \vdash e_2 \rightsquigarrow e'_2 \end{array}$$

$$\frac{\text{(EVAL}_{\rightsquigarrow})}{\mathcal{P} \vdash \text{eval}_{\ell} e \rightsquigarrow e''} \quad \begin{array}{l} \mathcal{P} \vdash e \rightsquigarrow e' \\ \mathcal{P} \vdash \text{parse}(s_j) \rightsquigarrow e_j \quad s_j \in \mathcal{P}(\ell) \quad x \text{ fresh} \\ e'' = \left( \begin{array}{l} \text{let } x = e' \text{ in} \\ \text{if } x \equiv s_1 \text{ then } e_1 \\ \text{else if } x \equiv s_2 \text{ then } e_2 \dots \\ \text{else safe\_eval}_{\ell} x \end{array} \right) \end{array}$$

$$\frac{\text{(SEND}_{\rightsquigarrow})}{\mathcal{P} \vdash e_0.\text{send}_{\ell}(e_1, \dots, e_n) \rightsquigarrow e'} \quad \begin{array}{l} \mathcal{P} \vdash e_i \rightsquigarrow e'_i \quad i \in 0..n \quad s_j \in \mathcal{P}(\ell) \quad x \text{ fresh} \\ e' = \left( \begin{array}{l} \text{let } x = e'_1 \text{ in} \\ \text{if } x \equiv s_1 \text{ then } e'_0.\text{parse}(s_1)(e'_2, \dots, e'_n) \\ \text{else if } x \equiv s_2 \text{ then } e'_0.\text{parse}(s_2)(e'_2, \dots, e'_n) \dots \\ \text{else safe\_eval}_{\ell} "e'_0." + x + "(e'_2, \dots, e'_n)" \end{array} \right) \end{array}$$

# TinyRuby Source Language

$e ::= x \mid v \mid d \mid e_1; e_2 \mid e_1 \equiv e_2 \mid \text{let } x = e_1 \text{ in } e_2$   
|  $\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \mid e_0.m(e_1, \dots, e_n)$   
|  $\text{eval}_\ell e \mid e_0.\text{send}_\ell(e_1, \dots, e_n)$   
|  $\text{safe\_eval}_\ell e \mid \llbracket e \rrbracket_\ell \mid \text{blame } \ell$   
 $v ::= s \mid \text{true} \mid \text{false} \mid \text{new } A \mid \llbracket v \rrbracket_\ell$   
 $d ::= \text{def}_\ell A.m(x_1, \dots, x_n) = e$

$x \in \text{local variable names}$      $A \in \text{class names}$   
 $m \in \text{method names}$              $s \in \text{strings}$   
 $\ell \in \text{program locations}$

# Properties of PRuby

## □ Translation Faithfulness:

**THEOREM 1** (Translation Faithfulness). *Suppose  $\langle \emptyset, \emptyset, e \rangle \rightarrow \langle M, \mathcal{P}', v \rangle$  and  $\mathcal{P}' \subseteq \mathcal{P}$  and  $\mathcal{P} \vdash e \rightleftharpoons e'$ . Then there exist  $M_{\mathcal{P}}, \mathcal{P}''$  such that  $\langle \emptyset, \emptyset, e' \rangle \rightarrow \langle M_{\mathcal{P}}, \mathcal{P}'', v \rangle$ .*

## □ Type Soundness:

**THEOREM 3** (Type Soundness). *If  $\emptyset \vdash e$  and  $\langle \emptyset, \emptyset, e \rangle \rightarrow \langle M, \mathcal{P}, r \rangle$ , then  $r$  is either a value or blame  $\ell$ . Thus,  $r \neq \text{error}$ .*

# Subset of TinyRuby

$e ::= x \mid v \mid d \mid e_1; e_2 \mid e_1 \equiv e_2 \mid \text{let } x = e_1 \text{ in } e_2$   
 $\mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \mid e_0.m(e_1, \dots, e_n)$   
 $\mid \text{eval}_\ell e \mid e_0.\text{send}_\ell(e_1, \dots, e_n)$   
 $\mid \text{safe\_eval}_\ell e \mid [e]_\ell \mid \text{blame } \ell$   
 $v ::= s \mid \text{true} \mid \text{false} \mid \text{new } A \mid [v]_\ell$   
 $d ::= \text{def}_\ell A.m(x_1, \dots, x_n) = e$

$x \in \text{local variable names}$       $A \in \text{class names}$   
 $m \in \text{method names}$               $s \in \text{strings}$   
 $\ell \in \text{program locations}$

## Converting paper proof to machine-based proof Coq

- Write language definitions in Coq
- Write profiling and transformation rules in Coq
- Write lemmas and theorems in Coq
- Finally Prove the lemmas and theorems in Coq!

# Language Definition

Definition  $\text{var} := \text{nat}$ .

Inductive  $\text{val} : \text{Set} :=$

- |  $\text{Str} : \text{string} \rightarrow \text{val}$
- |  $\text{True} : \text{val}$
- |  $\text{False} : \text{val}$ .

Definition  $\text{loc} := \text{nat}$ .

Inductive type :  $\text{Set} :=$

- |  $\text{tstr}$
- |  $\text{tbool}$
- |  $\text{tblame}$ .

Inductive  $\text{exp} : \text{Set} :=$

- |  $\text{RVar} : \text{var} \rightarrow \text{exp}$
- |  $\text{RVal} : \text{val} \rightarrow \text{exp}$
- |  $\text{RSeq} : \text{exp} \rightarrow \text{exp} \rightarrow \text{exp}$
- |  $\text{REq} : \text{exp} \rightarrow \text{exp} \rightarrow \text{exp}$
- |  $\text{RLet} : \text{var} \rightarrow \text{exp} \rightarrow \text{exp} \rightarrow \text{exp}$
- |  $\text{RIf} : \text{exp} \rightarrow \text{exp} \rightarrow \text{exp} \rightarrow \text{exp}$
- |  $\text{REval} : \text{loc} \rightarrow \text{exp} \rightarrow \text{exp}$
- |  $\text{RBlame} : \text{loc} \rightarrow \text{exp}$ .

# Variable Environment, Type Environment, and Profile

Definition vstate := var → val.

Definition tstate := var → type.

Definition profile := loc → list (prod string exp).

Definition override (X : Set) (f : nat → X) (k : nat) (x : X) :=  
 fun k' => if beq\_nat k k' then x else f k'.

Definition profile\_union (p1 : profile) (p2 : profile) : profile :=  
 fun l => let l1 := (p1 l) in  
 let l2 := (p2 l) in  
 list\_union l1 l2.

# Profiling Rules

Inductive profile\_op : vstate → exp → profile → exp → Prop :=

| PVALUE : forall st v,

profile\_op st (RVal v) empty\_profile (RVal v)

| PSEQ : forall st e1 e2 p1 p2 v1 v2,

profile\_op st e1 p1 v1 →

profile\_op st e2 p2 v2 →

profile\_op st (RSeq e1 e2) (profile\_union p1 p2) v2

.....

$$\frac{\text{(EVAL)} \quad \langle M, V, e \rangle \rightarrow \langle M_1, \mathcal{P}_1, s \rangle \quad \langle M_1, V, \text{parse}(s) \rangle \rightarrow \langle M_2, \mathcal{P}_2, v \rangle}{\langle M, V, \text{eval}_\ell e \rangle \rightarrow \langle M_2, (\mathcal{P}_1 \cup \mathcal{P}_2 \cup [\ell \mapsto s]), v \rangle}$$

Definition parse (e : exp) (s : string) : Prop :=

beq\_string (string\_of\_exp e) s = true.

.....

| PEVAL : forall st e e' l s p1 p2 v,

profile\_op st e p1 (Str s) →

parse e' s →

profile\_op st e' p2 v →

profile\_op st (REval l e) (profile\_add (profile\_union p1 p2) l (pair s e')) v

# Transformation Rules

Inductive  $\text{transform\_op} : \text{profile} \rightarrow \text{exp} \rightarrow \text{exp} \rightarrow \text{Prop} :=$

| TREFL\_VAR : forall p x,  
 transform\_op p (RVar x) (RVar x)  
 | TSEQ : forall p e1 e1' e2 e2',  
 transform\_op p e1 e1' →  
 transform\_op p e2 e2' →  
 transform\_op p (RSeq e1 e2) (RSeq e1' e2')

.....

| TEVAL: forall p l e e' e'',  
 transform\_op p e e' →  
 trans\_eval p (p l) e' e'' →  
 transform\_op p (REval l e) e''

with **trans\_eval** : profile → list (prod string exp) → exp → exp → Prop :=

| ENIL : forall p e' l,  
 trans\_eval p nil e' (RBlame l)  
 | ECONS : forall p pl e' e'' sj ej ej' ,  
 parse ej sj →  
 transform\_op p ej ej' →  
 trans\_eval p pl e' e'' →  
 trans\_eval p ((sj,ej)::pl) e' (RIf (REq e' (RVal (Str sj))) ej' e'').

(EVAL<sub>↔</sub>)

$$\mathcal{P} \vdash e \rightsquigarrow e'$$

$$\mathcal{P} \vdash \text{parse}(s_j) \rightsquigarrow e_j \quad s_j \in \mathcal{P}(l) \quad x \text{ fresh}$$

$$e'' = \left( \begin{array}{l} \text{let } x = e' \text{ in} \\ \text{if } x \equiv s_1 \text{ then } e_1 \\ \text{else if } x \equiv s_2 \text{ then } e_2 \dots \\ \text{else safe\_eval}_\ell x \end{array} \right)$$

$$\mathcal{P} \vdash \text{eval}_\ell e \rightsquigarrow e''$$

# Current progress

- Defined a reduced version of TinyRuby language in Coq
- Defined profile and transformation rules in Coq
- Formulated and partially proved translation faithfulness
  - ▣ Having trouble proving the translation faithfulness theorem, particularly  $\text{EVAL} \sim >$
- Formulated and partially proved type soundness

# Future Work



- Use more appropriate data type for profile like FSetList in Coq standard library
- Prove translation faithfulness and type soundness on the full TinyRuby language, which includes “send” and “method definition”

# References

- M. Furr, J. An, J. Foster, Profile-Guided Static Typing for Dynamic Scripting Languages
- Eduardo Giménez, Pierre Castéran, A Tutorial on [Co-]Inductive Types in Coq
- Diamondback Ruby  
(<http://www.cs.umd.edu/projects/PL/druby/>)
- Ruby  
([http://en.wikipedia.org/wiki/Ruby\\_\(programming\\_language\)](http://en.wikipedia.org/wiki/Ruby_(programming_language)))
- Coq (<http://www.lix.polytechnique.fr/coq/>)