Specification of RuScript

Zhen Zhang
Find me on GitHub @izgzhen

November 6, 2015
Draft v1.0

Contents

1 Abstract Computation Model

1.1 Representation

The source code is an array of instructions coded in binary. The running program is a stack of frames and object pool, in which every frame is composed of:

1. Local Stack (abbr. *stack*)
2. Local registers (abbr. *locals*)
3. Global registers (abbr. *globals*)
4. Frame code block (abbr. *code*)

Object pool is a managed heap. All living objects will live in heap, and accessible by `Gc` type, or *pointer*; when some object becomes inaccessible, it will be garbage collected.

The stack and registers are all container of pointers. The stack is where most computations operate on. The registers are meant for fast access of objects.

Stack can only be pushed or popped. It is assumed to be unlimited in size. And if stack underflowed, program will panic. Registers is an array of finite and fixed length. When the index is invalid, program will panic.

1.2 Code Structure

The grammar

```
<toplevel>      := [<statement>;]

<assignment>    := <identifier> = <expr>

<expr>          := <term> + <term>
                 | <term>

<term>          := <int>
                 | <ident>
                 | <string>
                 | new <ident>
                 | <ident>.<ident>([<expr>,])

<statement>     := <assignment>
                 | <classDecl>
                 | print <expr>
                 | return <expr>
```

```
<classDecl>     := class <ident> { [ <attrDecl> | <methodDecl> ] }

<attrDecl>      := <ident>

<methodDecl>    := fn <ident>([<ident>,]) {
                             [<globalDecl>]
                             [<statments>]
                         }

<globalDecl>    := global <ident>
```

Note #1: When we say "*A is defined in scope X*",

- If A is a variable, we mean that there exists an assignment `A = <expr>` in X
- If A is a method, we mean that A is defined in some class as a method
- If A is a class, we mean that A is defined in some top-level statement

Note #2: By "*variable*", we mean named primitive objects or instances

1.3 Scope and Scoping

Kinds of scope:

- Global (or top-level) scope: contains defined variables, classes
- Local (or method) scope: contains arguments, declared globals, variable defined in method source
- Class scope: contains only and always `self`. `self` can be used to refer the being-defined class's attributes and methods with common accessor pattern.

Scoping rules:

- Globals: Visible to top-level code after it is defined on top-level; Visible to method code after it is defined on top-level and declared as `global` at the beginning of method.
- Locals: Only visible to method's code
- Classes: Visible after defined

Note: The principle is to give warnings of name collision, and give errors when scoping found nothing. If collision happens, the scoping priorities are "local > class > global".

1.4 Object System

Current object system is rather simple. It has the following features:

- Everything is an *object*, and every computation is *message passing*
- *Class* can be instantiated to *instance*
- *Accessor* pattern, which we use to read *attributes* and invoke *method*

The `_Object` contains all kinds of object defined:

- Primitive Objects
  - `Int_ty`: 32-bit signed integer
  - `Array_ty`: Integer-indexed growable array of objects
  - `String_ty`: Immutable string
- `Frame_ty`: Representing a frame
- `Instance_ty`: Represent a general object as a instance of class
- `Class_ty`: Represent class definition

Every object must fulfill `Object` trait:

```
trait Object {
    fn call(&self,
            name: &str,
            args: Vec<Gc<_Object>>,
            env: &Gc<Env>,
            globals: &mut Vec<Gc<_Object>>) -> Gc<_Object>;

    fn tyof(&self) -> &str;
}
```

2 Stack System

Stack system is the implementation of the abstract computation model described above. It is an interpreter for variable-length instruction (i.e. *SCode*).

In the following text, all instructions and their operational semantics will be given in a formal way.

Note: Currently, the `SCode` is a mixture of classical data segment and text segment. All data will be carried in the instruction. But since actual binary layout has little to do with semantics, I will try to avoid referring to actual binary layout in formalization, but using $INST(D1, D2)$ to claim the relationship between operator and operands.

2.1 All instructions

- `PUSHG(Integer)`
- `PUSHL(Integer)`
- `PUSHA(Integer)`
- `POPG(Integer)`
- `POPL(Integer)`
- `POPA(Integer)`
- `ADD`
- `CALL(Integer, String, Integer)`
- `RET`
- `NEW(Integer)`
- `PUSH_INT(Integer)`
- `PUSH_STR(String)`
- `FRMEND`
- `CLASS(Integer, Integer)`
- `PRINT`

Naming conventions: `G` means global, `L` means local, `A` means attribute, `PUSHX(i)` means push `i`-indexed element of `X` on top of stack, `POPX(i)` means pop top of stack to `i`-indexed position of `X`.

2.2 Formal Computation Model

- Environment: E, tuple (L, G, S, C)
- Locals: L, mutable finite array
- Globals: G, mutable finite array
- Stack: S, mutable infinite stack
- Classes: C, immutable array of class definitions
- Code block: Σ, immutable array of instructions
- Instruction Space: I, set of defined instructions
- Frame: F, tuple (Σ, E)
- Object: ω, any entity satisfying `Object` trait
- Heap: H, set of managed objects
- Pointer: ψ, a managed reference to some $\omega \in H$
- Program: P, tuple of (C, Σ)
- Identifier Space: η
- Expression: e

Note: For commonly used collection type, set or array will be notated as [], tuple will be notated as ()

A complete computation is defined as a function $f_{comp} : (P, World) \to World'$, in which the $World'$ is temporarily defined as (B, H), in which B is standard output buffer.

Note: To make formula cleaner, we might use monad-like notation to describe side-effect, i.e. $f : A \to EnvM(B)$ is equivalent to $f : (A, Env) \to (B, Env')$

First, P will be loaded into a root frame: $f_{load} : P \to F_0$.

Then, we will define an undecidable computing process $f_{run} : F_i \to WorldM(\omega_i)$

If F is the root frame, then ω_0 will be discarded and program terminates, if not, ω_i will be *returned* to caller and the current frame will be popped out.

Method invoking is a way to create a frame, run it and turn the returned value into a *term*. It is a way to abstract computation. It is actually a special case of object `call` trait. $f_{invoke} : (\omega, m, [e]) \to WorldM(F)$. So by invoking, we can get value of call expression $f_{eval}(\omega.m([e]) = (f_{invoke}(\omega, m, [e]) \gg= f_{run})$

Thus, we can abstract the computation as a recursive process of "invoking, frame interpretation, finally returning value". Formally, we will present the "computation progress" as a stack of frames: $F_0 \diamond F_1 \diamond F_2 ... \diamond F_i$, where F_i is the current frame.s All $F_0 ... F_{n-1}$ is "suspended computation".

In summary, by a single step interpretation, we will have

- Frame computation: $F_0 \diamond F_1 \diamond F_2 ... \diamond F_i \to F_0 \diamond F_1 \diamond F_2 ... \diamond F_i'$
- Frame suspension and creation: $F_0 \diamond F_1 \diamond F_2 ... \diamond F_i \to F_0 \diamond F_1 \diamond F_2 ... \diamond F_i \diamond F_{i+1}$
- Frame continuation and completion: $F_0 \diamond F_1 \diamond F_2 ... \diamond F_i \to F_0 \diamond F_1 \diamond F_2 ... \diamond F_{i-1}$

Note that although we don't explicitly emphasize every time, but all computational transformation is accompanied with some side-effects, which is formalized with $WorldM$.

In the following formal semantics, if it is simply a frame computation, then we will conceal the frame structure; But if not, we will take everything into the formal presentation.

2.3 Formal Semantics

`PUSHG(Integer)`

$$\langle (L, G, C, S), PUSHG(i) \rangle \Rightarrow WorldM(L, G, C, G[i] \rhd S)$$
$$WorldM \rightsquigarrow H[G[i]] \uparrow$$

`PUSHL(Integer)`

$$\langle (L, G, C, S), PUSHL(i) \rangle \Rightarrow WorldM(L, G, C, L[i] \rhd S)$$
$$WorldM \rightsquigarrow H[L[i]] \uparrow$$

PUSHA(Integer)

$$\langle (L, G, C, S@(\omega \rhd S')), PUSHA(i) \rangle \Rightarrow WorldM(L, G, C, \omega.attrs[i] \rhd S)$$
$$WorldM \rightsquigarrow H[\omega] \uparrow$$

POPG(Integer)

$$\langle (L, G, C, \omega \rhd S'), POPG(i) \rangle \Rightarrow WorldM(L, subst(G, i, \omega), C, S')$$
$$WorldM \rightsquigarrow H[G[i]] \downarrow$$

POPL(Integer)

$$\langle (L, G, C, \omega \rhd S'), POPL(i) \rangle \Rightarrow WorldM(subst(L, i, \omega), G, C, S')$$
$$WorldM \rightsquigarrow H[L[i]] \downarrow$$

POPA(Integer)

$$\langle (L, G, C, \omega_1 \rhd \omega_2 \rhd S'), POPA(i) \rangle \Rightarrow WorldM(L, G, C, subst(\omega_2, i, \omega_1) \rhd S')$$
$$WorldM \rightsquigarrow H[\omega_2.attrs[i]] \downarrow$$

ADD

$$\langle (L, G, C, \omega_1 \rhd \omega_2 \rhd S'), ADD \rangle, \omega_3 \sim \omega_1 + \omega_2 \Rightarrow WorldM(L, G, C, \omega_3 \rhd S')$$
$$WorldM \rightsquigarrow H[\omega_1] \downarrow, H[\omega_2] \downarrow, H.new(\omega_3)$$

CALL(Integer, String, Integer)

$$\left.\begin{array}{l} \langle (L, G, C, \omega_1 \rhd \omega_2 \rhd ... \omega_n \rhd S'), CALL(i, m, n) \rangle \\ \omega \leftarrow scope(L + G, i), \omega.m(\omega_1, ..., \omega_n) \sim \omega' \end{array}\right\} \Rightarrow WorldM(L, G, C, \omega' \rhd S')$$
$$WorldM \rightsquigarrow \forall i \in 1...n, H[\omega_i] \downarrow, H.new(\omega')$$

RET

$$F_0@\langle (L, G, C, \omega \rhd S'), RET \rangle \Rightarrow terminate$$
$$F_0 \diamond ... \diamond F_{i-1}@(Susp : \omega \rightarrow \langle E, i \rangle) \diamond F_i@\langle (L, G, C, \omega \rhd S'), RET \rangle \Rightarrow F_0 \diamond ... \diamond F_{i-1}@\langle E, i \rangle$$

NEW(Integer)

$$\langle (L, G, C, S), NEW(i) \rangle, \omega \leftarrow C[i].new() \Rightarrow WorldM(L, G, C, \omega \rhd S)$$
$$WorldM \rightsquigarrow H.new(\omega)$$

`PUSH_INT(Integer)`

$$\langle (L, G, C, S), PUSH\_INT(i) \rangle, \omega \leftarrow Int.new(i) \Rightarrow WorldM(L, G, C, \omega \rhd S)$$
$$WorldM \rightsquigarrow H.new(\omega)$$

`PUSH_STR(String)`

$$\langle (L, G, C, S), PUSH\_INT(s) \rangle, \omega \leftarrow String.new(s) \Rightarrow WorldM(L, G, C, \omega \rhd S)$$
$$WorldM \rightsquigarrow H.new(\omega)$$

`FRMEND` This is a pseudo-instruction.

`CLASS(Integer, Integer)` This is a pseudo-instruction.

`PRINT`

$$\langle (L, G, C, \omega \rhd S), PRINT \rangle \Rightarrow WorldM(L, G, C, \omega \rhd S)$$
$$WorldM \rightsquigarrow B.print(\omega)$$

3 Exception handling and correctness protocol

3.1 Exception Handling

Currently, RuScript doesn't have error handling capacity, if any undefined behaviour happens, the interpreter will simply panic.

However, a protocol of correctness should exist. First, the language should have only high-level exceptions, like "divide by zero", "no such method", "out of index" and so on, but never low-level errors caused by virtual machine itself, like stack underflow, register indexing error, invalid instruction et cetera. To be able to include the first category of exceptions in the specification, we should make sure that they can't cause interpreter panic at runtime. The defined exceptions should print debugging information and exit normally. Such a protocol should be implemented by compiler.

> Correctness Protocol: *As long as runtime system is enforcing the formal model, any compiled code should not cause panic to runtime system*

3.2 Correctness Protocol Implementation

Essentially, this implementation is a *correctness proof of an existing compiler*. So, here we will only give some examples of implementation.

The hand-written proofs can be lengthy, imprecise and unreadable, so maybe it is possible to construct a model in Coq.

Example Prove that the following excerpt can generate correct code for expression $t_1 + t_2$, assuming that `pushTerm` can emit a segment of code, which when executed, can correctly leave the computed value on top of stack.

```
pushExpr (Plus tm1 tm2) = do
    pushTerm tm1
    pushTerm tm2
    emit SAdd
```

Proof:

According to assumption, let `pushTerm` be such a function $f_{term} : Term \to CompilerM(\Sigma)$, in which $Term$ is lexical representation of a term, and $CompilerM$ is the monad analogy of a compiler. And for any $t \in Term$, if t is computable under environment E, then $\langle E, \Sigma \rangle \Rightarrow E'$, in which top of stack of E' is value-equivalent of $Term$.

So, let

$$\Sigma_1 \leftarrow f_{term}(t_1), \Sigma_2 \leftarrow f_{term}(t_2)$$

then

$$\Sigma_1 + \Sigma_2 + [SAdd] \leftarrow f_{expr}(Plus(t1, t2))$$

since

$$\langle (L, G, C, S), \Sigma_1 \rangle, \omega_1 \sim t_1 \Rightarrow (L, G, C, \omega_1 \rhd S)$$
$$\langle (L, G, C, S), \Sigma_2 \rangle, \omega_2 \sim t_2 \Rightarrow (L, G, C, \omega_2 \rhd S)$$

so

$$\langle (L, G, C, S), \Sigma_1 + \Sigma_2 \rangle, \omega_1 \sim t_1, \omega_2 \sim t_2 \Rightarrow (L, G, C, \omega_2 \rhd \omega_1 \rhd S)$$

then

$$\langle (L, G, C, S), \Sigma_1 + \Sigma_2 + SAdd \rangle, \omega_3 \sim (t_1 + t_2) \Rightarrow (L, G, C, \omega_3 \rhd S)$$

which conforms to the spec.