The Essence of Ruby[*]

Katsuhiro Ueno, Yutaka Fukasawa[**], Akimasa Morihata[***], and Atsushi Ohori

Research Institute of Electric Communication, Tohoku University
{katsu, fukasawa, morihata, ohori}@riec.tohoku.ac.jp

Abstract. Ruby is a dynamic, object-oriented language with advanced features such as `yield` operator and dynamic class manipulation. They make Ruby a popular, highly productive scripting language, but they also make the semantics of Ruby complicated and difficult to understand. Even the JIS/ISO standard of Ruby seems to contain some ambiguities. For Ruby to be established as a reliable scripting language, it should have a rigorous semantics. To meet this challenge, we present a formal operational semantics that can serve as a high-level specification for both the users and implementers. The key insight underlying the semantics is that various elaborate features of Ruby can be cleanly represented as a composition of two orthogonal calculi: one for objects and classes and the other for representing control. The presented semantics leads straightforwardly to a reference implementation. Initial evaluation of our implementation confirms that the presented semantics conforms to commonly accepted Ruby behavior.

Keywords: Ruby, operational semantics, iterator, dynamic language

1 Introduction

Ruby is a dynamic object-oriented language. Its design rationale is to provide the programmer a language that is easy and natural to write programs with maximum freedom. Its rich set of control abstractions such as *iterator* with `yield` operator enable the programmer to write sophisticated programs succinctly and efficiently, and its flexible dynamic semantics allows the programmer to configure evaluation environment including classes and their method suites. These features make Ruby a popular, highly productive scripting language. These features, however, make the semantics of Ruby complicated and difficult to understand. Clear and formal understanding of the semantics is essential for reasoning, verification, and thereby developing reliable software.

To address this issue, the designers and the user communities have been trying to define the standard of the language. The efforts have culminated to

[*] The first author has been partially supported by JSPS KAKENHI Grant Number 24700021. The first and fourth author has been partially supported by JSPS KAKENHI Grant Number 25280019.

[**] The second author's current affiliation: Marvelous Inc.

[***] The third author's current affiliation: The University of Tokyo.

the JIS (Japanese Industrial Standard) standard [10] of Ruby followed by the ISO standard [8]. We based our research on the original JIS standard [10] written in Japanese; the ISO counterpart [8] is essentially the same. This 226 pages document defines the syntax and the semantics of Ruby in details by defining global run-time states including 7 independent run-time stacks, and specifying how each syntax of Ruby mutates the run-time states.

Studying the details of this document, we find it admirable to have completed the detailed descriptions of the evaluation process of all the language constructs. It can certainly serve as a guideline for a particular style of Ruby implementations, including the widely used CRuby interpreter [15]. As a specification of the semantics of Ruby, however, there seem to contain certain amount of ambiguities in its descriptions of the interactions of some of the elaborate language features. As a concrete example, let us consider the following Ruby program.

```
def foo ; yield ; print "foo" ; end
def bar ; foo { yield } ; print "bar" ; end
bar { break }
print "end"
```

`yield` calls a block (a form of a lambda term written as `{···}`) passed as an argument to a method. The block `{ break }` will be called from the `yield` in the first line through nested `yield` evaluations. The effect of the entire code should be to reset the run-time environment to those just before the execution of the third line and to jump to the forth line, according to a common informal understanding of `yield` with `break`. So this program only prints "`end`". We have to say that the standard is ambiguous in specifying the precise meanings of such codes. The document suggests some complication in evaluation of (nested) `yield`s with control effects in its informal statement such as "possibly affected execution context", whose precise meanings remain to be clarified.

In principle, the specification could be refined to the point that the evaluation process is completely specified, but such a refinement would result in too detailed and too low-level descriptions to understand the semantics of Ruby, let alone to reason about it. A rigorous high-level specification of Ruby should ideally be structured as a simple composition of well-structured components, each of which is a straightforward specification of individual language feature. However, decomposing Ruby into such components is not trivial since Ruby's features are deeply interwoven with each other due to its dynamic nature with sophisticated control operations such as `yield`s.

To meet this challenge, in this paper, we construct a big-step operational semantics (in the style of natural semantics [11]) of Ruby based on our following observations: the features of Ruby can be categorized as two disjoint subsets, one for object managements and another for control operations, and each of the two categories can be defined as a separate simple term calculus by abstracting the intersections between the two calculi as oracle relations. The object counterpart is *the object calculus* that accounts for dynamic behavior of instance variables, method look-up, and class manipulation. The control counterpart is *the control calculus* that describes local variable binding, method invocation, and Ruby's blocks with global control effects.

Our approach is inspired by the work of Guha et. al. [7], where a single calculus of JavaScript is explained through its two aspects of object and control operations. A novel contribution of the present paper is to show that these two aspects are presented as independent two calculi and the entire calculus is obtained by composing the two. This approach of modeling a dynamic language by composition of multiple calculi would be beneficial to understanding its semantic structure.

Each calculus can be further decomposed into its essential core part and extensions to the core. Each small component of this decomposition represents individual Ruby feature with an effectively small syntax and evaluation rules. The two core calculi and their extensions are combined straightforwardly into the Ruby calculus that represents the essential core of Ruby. The real Ruby is obtained by introducing a syntax elaboration phase that translates a number of context-dependent implicit constructs into explicit terms in the Ruby calculus. The composition of the syntax elaboration and the semantics of the Ruby calculus defines the semantics of the real Ruby language. The resulting semantics serves as a high-level specification of Ruby. Since the big-step semantics directly corresponds to an interpreter implementation, it also straightforwardly yields a reference implementation in a functional language with generative exceptions. We have implemented the semantics in Standard ML and have conducted initial evaluation against a subset of Ruby test programs, which we believe cover typical cases of Ruby's dynamic features and unique control structures. The results confirm that the presented semantics with respect to blocks and `yield`s conforms to commonly accepted Ruby behavior.

In a wider perspective, the results shown in this paper can be regarded as a case study of developing a sophisticated dynamic language in a formal and clear way. Recent programming languages have complex control operators and flexible dynamic features which are considered practical but dirty. In addition, such languages evolve rapidly with drastic changes according to the needs of software developers. To prevent the specifications of such languages from becoming obsolete quickly, the construction of each specification as well as the specification itself must be clear and concise. Our developments shown in this paper suggest that this requirement can be satisfied by dividing their complex features into simple and nearly orthogonal subsets.

The rest of this paper is organized as follows. Section 2 overviews the syntax of Ruby and describes our strategy in defining the operational semantics. Section 3 defines the two calculi with their operational semantics, and combines the two calculi to form the essential core of Ruby. Section 4 extends the two calculi to represent the full functionality of Ruby. Section 5 defines the Ruby calculus. Section 6 describes the syntactic elaboration from real Ruby to the Ruby calculus. Section 7 describes our implementation and reports its evaluation results against crucial test cases. Sections 8 discusses related works. Sections 9 concludes the paper. The Appendix contains the definitions of the calculi presented in this paper.

2 Overview of Ruby and our strategies

Ruby is a class-based “pure” object-oriented language, where all data elements are objects, and all the computations are done through method invocations on receiver objects. For example, `1 + 2` invokes the `+` method on an instance `1` of class `Integer` with an argument object `2`.

Each object belongs to a single class. A class itself is also an object, which maintains a mutable set of methods for its instance objects. The system maintains the class hierarchy tree induced by the inheritance relation. Each object has identity and a set of mutable instance variables. Instance variables are dynamically generated in each object when they are initially assigned to some value. The following is a simple example of cons cell for lists.

```
class Cons                            # define "Cons" class.
  def setcar(x) ; @car = x ; end      # define "setcar" method in Cons.
  def setcdr(x) ; @cdr = x ; end      # define "setcdr" method in Cons.
  def car ; @car ; end                # define "car" method to Cons.
  def cdr ; @cdr ; end                # define "cdr" method to Cons.
end
```

`@car` is an instance variable. When `setcar` is called on a `Cons` object at the first time, this instance variable is generated in the receiver object.

Another feature that distinguishes Ruby from other object-oriented languages is its control mechanism called *block*, which is a piece of code that takes a parameter and returns a value. A block is passed as an optional parameter to a method. The passed block is invoked through an expression of the form `yield(`e`)` occurring in the body of the method. This expression *yields* the value denoted by e to the invoked block i.e. evaluates the block with the value. As an example, suppose we want to enrich the `Cons` class by adding an enumeration method `each`. With `yield`, this can be done by writing the following code just after the previous definitions of `Cons` class.

```
class Cons
  def each                                # add "each" method to Cons.
    yield(@car)                           # call the given block.
    @cdr.each { |x| yield(x) } if @cdr    # call "each" with a block.
  end
end
```

Since class and method definitions are statements whose effects are mutations to the class object, this code adds `each` method to the `Cons` class previously defined. This `each` method takes a block implicitly and calls the block with a cons cell value by the `yield(@car)` expression. Then it calls itself for the following cons cells with a block `{ |x| yield(x) }` which propagates the given block to the recursive call. As a result, `each` method calls the given block with each cons cell value. For example, if `l` is an instance of `Cons` then `l.each { |x| print x }` prints each elements in `l`.

So far, blocks act just like lambda abstractions. The real strength of Ruby comes from the combination of blocks and a rich collection of non-local control statements such as `break` that breaks out of either an iteration or a block. This feature allows users to define various iteration abstractions in a concise and intuitive manner, as witnessed in the left-hand code below, which has the same structure and the intuitive meaning as the corresponding imperative one on the right:

using the `each` iterator:

```
sum = 0
l.each { |i|
  break if i == 0
  sum = sum + i
}
```

using `while` loop:

```
sum = 0
i = l
while i do
  break if i.car == 0
  sum = sum + i.car
  i = i.cdr
end
```

Both the above programs accumulate values in the cons list `l` to variable `sum` until they meet zero. In both programs, `break` is used to terminate the iteration on `l` regardless of the form of the loop. Thanks to this feature, skillful Ruby programmers can write high-level object-oriented codes in a natural, intuitive, and efficient way.

As overviewed above, major features of Ruby are classified into two categories. The first is the *object structure* to represent dynamic behavior of objects and classes. The second is the *control structure* to support blocks with non-local control. The key observation underlying our construction of the semantics of Ruby is that these two structures can be cleanly represented by two independent sub-calculi, which we call *the object calculus* and *the control calculus.* The object calculus accounts for various dynamic features of objects including instance variables and dynamic method bindings. The control calculus realizes the mechanism of `yield`-ing a value to a block with a rich set of non-local control statements including `break`. This calculus accounts for Ruby's subtle control flow in a simple and precise manner.

After this separation, the object calculus can be defined relatively easily as a variant of dynamic object calculi extensively studied in literature (see [1] for a survey). In contrast, defining the control calculus requires to describe control flows that the combination of blocks and non-local jumps produce. James and Sabry [9] proposed a continuation-based approach to describe a generalized `yield` operator. However, continuation is not preferable to reveal the essential core of Ruby, which must looks like a subset of Ruby language, since it does not directly correspond to the commonly understood Ruby's control primitives and the Ruby implementations.

Our observation is that ML-style generative exception provides just enough power to represent these non-local jump statements. ML-style exceptions are generative in the sense that an exception definition dynamically generates a fresh exception tag every time it is evaluated at runtime, in contrast to other statically typed languages, such as C++ and Java, which uses statically declared types or classes as exception tags. The following example written in Standard ML shows the characteristics of the generative exceptions.

```
fun f () = let exception E
           in (fn () => raise E, fn g => g () handle E => ())
           end
val (r1, h1) = f ()
and (r2, h2) = f ()
val _ = h1 r2;  (* this causes uncaught exception *)
```

It defines function `f`, which creates a pair of functions that raises and handles exception `E`, and then calls `f` twice to obtain four functions: `r1` and `r2` which raise `E`, and `h1` and `h2` which handle `E`. Although they seem to deal with identical exception `E`, `h1` handles only exceptions raised by `r1` since two calls of `f` generates different exception tags for `E`. So function application `h1 r2` causes uncaught exception error.

By including generative exception tags in the evaluation context, the evaluation relation precisely models Ruby's subtle behavior. Generativity is essential. To see this, consider the following example.

```
def foo(x)
  print "E#{x} "
  if x >= 5 then yield
  else foo(x+1) { print "B#{x} " ; if x <= 2 then break else yield end }
  end
  print "L#{x} "
end
foo(1)
```

Method `foo` recursively calls itself until `x` becomes `5`, at which point the block is called from the `yield` statement in the body of `foo` method. The called block then executes `yield` inside of the block, resulting in calling the blocks of the previous invocations of `foo` until the one with the parameter 2. This block executes `break` whose effect is to jump to the statement `print "L#{x} "` of the method body of `foo` at the method invocation `foo(2)`. As a result, this code produces the following output: `E1 E2 E3 E4 E5 B4 B3 B2 L2 L1`. As seen in this example, destinations of `break`s cannot be statically labeled. This situation can be cleanly represented by generating a new exception tag for each execution of method call with a block and handling the generated exception.

By carefully constructing the two calculi in such a way that each calculus treats the features that are external to the calculus as oracles (primitives), the composition of the two calculi can be trivially done, yielding the Ruby calculus representing the essence of Ruby. By introducing syntax elaboration and various primitive classes, the combined calculus extends to full Ruby.

3 The essential core of Ruby

Following the strategy described in the previous section, we define two sub-calculi, the object calculus for object management, and the control calculus for control flows. These two sub-calculi are orthogonal except method invocation that is the point they intersect with each other. The intersection point of two calculi is explicitly specified by a pair of oracle relations. By combining two sub-calculi at the intersection point, the Ruby calculus is obtained. In this section, we define the essential core part of two calculi and combine them into the core Ruby calculus, which captures fundamentals of Ruby.

To define the calculi, we introduce some notations. $\{\}$ is an empty map. $\mathrm{dom}(F)$ is the domain of a function F. For a function F, $F \oplus \{x \mapsto v\}$ is the function F' such that $\mathrm{dom}(F') = \mathrm{dom}(F) \cup \{x\}$, $F'(x) = v$, and $F'(y) = F(y)$ for any $y \in \mathrm{dom}(F')$ such that $x \neq y$. We also define $F \oplus G$ by extending the definition of $\oplus$ to any function G. $F \ni \{x \mapsto v\}$ indicates that $x \in \mathrm{dom}(F)$ and $F(x) = v$. $[\,]$ is empty list and $[x]$ is a singleton list of an item x. $l_1 +\!\!+\, l_2$ is the concatenation of two lists l_1 and l_2. We use list comprehension of the form $[F(x) \mid x \in l, P(x)]$ to generate a list in order from input list l where $P(x)$ is a predicate on an input item x and $F(x)$ is the output item corresponding to x. We define a big operator on lists $\bigoplus$ that fold the given list of maps by $\oplus$.

3.1 The core object calculus

We let k, i and m range over the given set of *class identifiers*, *instance variable identifiers*, and *method identifiers*, respectively. We assume that there is a given set of *method entities* ranged over by ξ. In the object calculus, it is enough to treat a method entity as an atomic language construct.

The syntax of the core object calculus is given below.

$$e ::= k \mid e.i \mid e.i\ \texttt{=}\ e \mid e.m \mid \texttt{alloc}\ e \mid \texttt{new\_class}\ e \mid \texttt{def}\ e\texttt{\#}m\ \texttt{=}\ \xi$$

$e.i$ accesses instance variable i. $e_1.i$ = e_2 assigns object e_2 to instance variable i of object e_1. They require to specify the object e to which i belongs, while in Ruby it is always the object bound to `self` and omitted. $e.m$ looks up method m of an object e. We note that this calculus focuses only on object operations, and does not define how methods are invoked and how argument values are passed. `alloc` e allocates an instance object of class e, which corresponds to `Class#new` method of Ruby. `new_class` e dynamically creates a new direct subclass of a class e. `def` e`#`m `=` ξ binds m to ξ in class e.

For this calculus, we define semantic structures necessary to evaluate this calculus as follows.

A class consists of its method binding and its direct super class. Let $\mathcal{M}$ range over *method bindings*, which are finite functions form method names to method entities. An *instance method context*, ranged over by M, is a finite function from class identifiers to method bindings. A *class inheritance context*, ranged over by K, is a finite function over class identifiers that associates a class to its direct super class. We refers to a pair (K, M) of the above two contexts as *class context* ranged over by $\mathcal{K}$. We say that K is well-formed if the transitive closure of K is irreflexive. This well-formedness condition means that no class inherits itself. We define *ancestors of k in K*, referred to as *ancestors*(K, k), as follows.

$$\mathit{ancestors}(K,k) = \begin{cases} \epsilon & \text{if } k \notin \mathrm{dom}(K) \\ \mathit{ancestors}(K,k') +\!\!+ [k] & \text{if } K \ni \{k \mapsto k'\} \end{cases}$$

Note that $\mathit{ancestors}(K, k)$ is the list of the ancestor classes in the reverse order, i.e. the front of the list is the farthest ancestor. If K is well-formed, the ancestors list of any k in K is always finite. As we shall see, all the rules we shall define preserve well-formedness of K. We define function $\mathit{methods}(\mathcal{K}, k)$ that computes the map of available instance methods of class k in $\mathcal{K}$ as follows.

$$\mathit{methods}((K, M), k) = \bigoplus \big[M(x) \bigm| x \in \mathit{ancestors}(K, k) \big]$$

In Ruby, there are several built-in classes that require special constructors to allocate those instances. To prevent allocating instances of such classes by the generic constructor `alloc`, we define *Special* as the set of the class identifiers of such built-in classes. we let *Special* be $\{\texttt{Class}, \texttt{Module}, \texttt{Proc}\}$ in this paper.

Let v range over the given countably infinite set of *object identifiers*, or *values.* As we mentioned, all data including classes are objects in Ruby, so a Ruby expression always denotes an object identifier. We let a range over the set of non-class object identities. So v is either k or a.

Any object in Ruby has its own instance variables and belongs to a class. Let $\mathcal{I}$ range over *instance variable bindings*, which are finite functions from instance variable names to object identifiers. An *object entity* is a pair $(\mathcal{I}, k)$ of an instance variable binding $\mathcal{I}$ and its class k. An *object heap*, ranged over by I, is a finite function from object identifiers to object entities.

The operational semantics of the object calculus is defined as a set of rules to derive the *evaluation relation* of the form $I, \mathcal{K} \vdash e \Downarrow r, I', \mathcal{K}'$ indicating the fact that, under the context I and $\mathcal{K}$, an expression e evaluates to a result r, and produces a new context I' and $\mathcal{K}'$. We note that due to the dynamic nature of Ruby, a class context $\mathcal{K}$ also acts as another mutable store in addition to an object heap I. A result r is either a value v or *wrong*. *wrong* indicates run-time failure.

As we mentioned above, we do not specify the structure of method entity in this calculus. To define a rule for method look-up term $e.m$, we assume that there is an oracle evaluation relation $I, \mathcal{K} \vdash \xi \leadsto r, I', \mathcal{K}'$ to convert a found method entity ξ to a result r and to return a new context I' and $\mathcal{K}'$ under the context I and $\mathcal{K}$. This oracle shall be replaced by the control calculus we shall define below.

Figure 1 shows the set of evaluation rules that derives the evaluation relation. We note that the above set of rules, and all the rules we shall define in the sequel, should be taken with the following implicit rules leading *wrong*: if any of the conditions in the premises are not satisfied, or evaluation of any component of a term yields *wrong*, then the entire evaluation will yield *wrong*.

3.2 The core control calculus

Let x and b range over a given countable set of *local variable identifiers* and *block identifiers*, respectively. The set of the control calculus is given by the following syntax.

$$e ::= \texttt{bind}\ x\ \texttt{=}\ e\ \texttt{in}\ e \ \mid\ x \ \mid\ \texttt{update}\ x\ \texttt{=}\ e \ \mid\ \texttt{proc}\ \{\, |x|\ e\, \}\ \texttt{as}\ b\ \texttt{in}\ e$$

$$\textsc{class}\ \frac{(K, M) = \mathcal{K} \quad k \in \mathrm{dom}(K) \cap \mathrm{dom}(M)}{I, \mathcal{K} \vdash k \Downarrow k, \mathcal{K}}$$

$$\textsc{ivar}\ \frac{I, \mathcal{K} \vdash e \Downarrow v, I', \mathcal{K}' \quad I' \ni \{v \mapsto (\mathcal{I}, k)\} \quad \mathcal{I} \ni \{i \mapsto v'\}}{I, \mathcal{K} \vdash e.i \Downarrow v', I', \mathcal{K}'}$$

$$\textsc{istore}\ \frac{I, \mathcal{K} \vdash e_1 \Downarrow v_1, I', \mathcal{K}' \quad I', \mathcal{K}' \vdash e_2 \Downarrow v_2, I'', \mathcal{K}'' \quad I'' \ni \{v_1 \mapsto (\mathcal{I}, k)\}}{I, \mathcal{K} \vdash e_1.i\ \texttt{=}\ e_2 \Downarrow v_2, I'' \oplus \{v_1 \mapsto (\mathcal{I} \oplus \{i \mapsto v_2\}, k)\}, \mathcal{K}''}$$

$$\textsc{call}\ \frac{\begin{array}{c} I, \mathcal{K} \vdash e \Downarrow v, I', \mathcal{K}' \quad I' \ni \{v \mapsto (\mathcal{I}, k)\} \quad \mathit{methods}(\mathcal{K}', k) \ni \{m \mapsto \xi\} \\ I', \mathcal{K}' \vdash \xi \leadsto r, I'', \mathcal{K}'' \end{array}}{I, \mathcal{K} \vdash e.m \Downarrow r, I'', \mathcal{K}''}$$

$$\textsc{alloc}\ \frac{I, \mathcal{K} \vdash e \Downarrow k, I', \mathcal{K}' \quad k \notin \mathit{Special} \quad a \text{ fresh}}{I, \mathcal{K} \vdash \texttt{alloc}\ e \Downarrow a, I' \oplus \{a \mapsto (\{\}, k)\}, \mathcal{K}'}$$

$$\textsc{newClass}\ \frac{I, \mathcal{K} \vdash e \Downarrow k', I', (K', M') \quad k \text{ fresh}}{\begin{array}{r} I, \mathcal{K} \vdash \texttt{new\_class}\ e \Downarrow k, I' \oplus \{k \mapsto (\{\}, \texttt{Class})\}, \\ (K' \oplus \{k \mapsto k'\}, M' \oplus \{k \mapsto \{\}\}) \end{array}}$$

$$\textsc{def}\ \frac{I, \mathcal{K} \vdash e \Downarrow k, (K', M') \quad M' \ni \{k \mapsto \mathcal{M}\}}{I, \mathcal{K} \vdash \texttt{def}\ e\texttt{\#}m\ \texttt{=}\ \xi \Downarrow k, I, (K', M' \oplus \{k \mapsto \mathcal{M} \oplus \{m \mapsto \xi\}\})}$$

Fig. 1. Evaluation rules of the object calculus

$$\begin{array}{rl} & \mid\ \texttt{yield}\ e\ \texttt{to}\ b\ \mid\ e.m(e, \texttt{\&}b)\ \mid\ e; e\ \mid\ j\ e \\ j ::= & \texttt{return}\ \mid\ \texttt{break}\ \mid\ \texttt{next} \end{array}$$

`bind` x `=` e_1 `in` e_2 introduces local variable binding x in e_2. `update` x `=` e destructively assigns value e to x. In Ruby, the standard syntax for assignments `x = e` is overloaded with variable definition (corresponding to `bind`) or variable update (corresponding to `update`), dependent on its context. As we shall mention in Section 6, this overloaded syntax can be supported by syntactic elaboration. `proc` $\{\,|x|\ e_1\,\}$ `as` b `in` e_2 binds b to block $\{\,|x|\ e_1\,\}$ in e_2. `yield` e `to` b calls block b with an argument e. $e_1.m(e_2, \texttt{\&}b)$ calls method m of object e_1 with an argument e_2 and a block b. e_1; e_2 is sequential execution. $j\ e$ performs non-local jumps of the three kinds. `return` e breaks out of the currently executing method. `break` e breaks out of the method invocation with the currently executing block. `next` e breaks out of the currently executing block. We refer to j as a *jump kind* in what follows. Ruby syntactically prohibits some of meaningless `break`s. For simplicity, we do not introduce syntax restriction here. Instead, meaningless `break` is represented as *wrong*.

We define the semantic structure to evaluate this calculus as follows.

Non-local jumps performed by $j\ e$ are realized by meta-level exceptions. We follow a Standard ML style approach in representing exceptions [13], which is well studied in literature. Let t range over a given countably infinite set of (meta-level) *exception tags*. Let T be an *exception context*, which is a finite function from jump kinds to exception tags. A *packed value* $[v]^t$ is a pair of value v and an exception tag t, representing a raised exception with tag t and a parameter value v. The *result*, ranged over by r, is either a value v, packed value $[v]^t$, or *wrong*.

Different from the object calculus, a method entity ξ is defined to be a triple (x, b, e) of a variable x and a block b for formal parameters and an expression e for the method body. To represent a mutable variable, we introduce a *variable store* ranged over by S. Let s range over a given countably infinite set of *variable references*. A variable store S is a finite function from variable references to values. In the control calculus, we treat a value v as an atomic entity. A *variable environment* E is a finite function from variables to variable references. A *block entity* ranged over by ρ is a tuple (E, T, x, e) consisting of evaluation contexts E and T, a formal parameter variable x, and a block body e. Let B range over *block contexts*, which are finite maps from block identifiers to block entities.

The evaluation relation is defined by a set of rules to derive either of the forms $S, B, E, T \vdash e \Downarrow r, S'$ or $S, B, E, T \vdash e$ handle $t \Downarrow r, S'$. The former is the standard form indicating the fact that under the evaluation context S, B, E and T, e evaluates to r and produces S'. In this case, an exception raised during the evaluation of e is propagated. The latter catches exceptions with tag t raised during the evaluation of e.

To define a rule for method invocation term $e_1.m(e_2, \&p)$ in this calculus, we use an oracle relation $S, B, E, T \vdash e.m \leadsto \xi, S'$ to look up a method entity specified by $e.m$, in contrast to the object calculus which uses oracle to invoke methods. This oracle shall be replaced by the object calculus.

Figure 2 shows the set of the evaluation rules. This set of rules should be taken with implicit rules for *wrong* as before. In addition, each rule in the standard form comes with an exception propagation rule: if a component yields an unexpected raised exception then cancel any subsequent evaluation of other components and the entire term will yield the raised exception. The top-level evaluation does not expect any raised exception.

3.3 The core Ruby calculus

The object calculus and the control calculus are mostly orthogonal, and merges straightforwardly into the core Ruby calculus, representing the essence of Ruby; the semantics of all language constructs of Ruby shall be described as either extensions or syntactic elaborations to the Ruby calculus.

In the Ruby calculus, the contexts I and $\mathcal{K}$ of the object calculus and the context S of the control calculus are all mutable stores. We let H range over *Ruby heaps*, which is a tuple of of I, $\mathcal{K}$ and S. The notation $H \oplus \{x \mapsto y\}$ and $H \ni \{x \mapsto y\}$ affect the corresponding part of H. We also naturally extend the function *methods*$(\mathcal{K}, k)$ to *methods*(H, k).

The evaluation relation of the merged calculus is of the form $H, B, E, T \vdash e \Downarrow r, H'$. The evaluation rules for the Ruby calculus are obtained by changing the rules so that each rules preserves non-attractive parts of given heap H in the returned heap. The only exception is the case for CALL. The case of CALL in the core Ruby calculus is obtained by combining two CALL rules as if each

$$\textsc{bind}\quad \frac{\begin{array}{c} S, B, E, T \vdash e_1 \Downarrow v, S' \\ S' \oplus \{s \mapsto v\}, E \oplus \{x \mapsto s\}, T \vdash e_2 \Downarrow r, S'' \ (s \text{ fresh}) \end{array}}{S, B, E, T \vdash \texttt{bind}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \Downarrow r, S''}$$

$$\textsc{var}\quad \frac{E \ni \{x \mapsto s\} \qquad S \ni \{s \mapsto v\}}{S, B, E, T \vdash x \Downarrow v, S}$$

$$\textsc{update}\quad \frac{E \ni \{x \mapsto s\} \qquad S, B, E, T \vdash e \Downarrow v, S'}{S, B, E, T \vdash \texttt{update}\ x\ \texttt{=}\ e \Downarrow v, S' \oplus \{s \mapsto v\}}$$

$$\textsc{proc}\quad \frac{\begin{array}{r} S, B \oplus \{b \mapsto (E, T \oplus \{\texttt{break} \mapsto t\}, x, e_1)\}, E, T \vdash e_2 \text{ handle } t \Downarrow v, S' \\ (t \text{ fresh}) \end{array}}{S, B, E, T \vdash \texttt{proc}\ \{\ |x|\ e_1\ \}\ \texttt{as}\ b\ \texttt{in}\ e_2 \Downarrow v, S'}$$

$$\textsc{yield}\quad \frac{\begin{array}{r} S, B, E, T \vdash e \Downarrow v, S' \qquad B \ni \{b \mapsto (E', T', x, e)\} \\ S' \oplus \{s \mapsto v\}, \emptyset, E' \oplus \{x \mapsto s\}, T' \oplus \{\texttt{next} \mapsto t\} \vdash e \text{ handle } t \Downarrow r, S'' \\ (s, t \text{ fresh}) \end{array}}{S, B, E, T \vdash \texttt{yield}\ e\ \texttt{to}\ b \Downarrow r, S''}$$

$$\textsc{call}\quad \frac{\begin{array}{r} S, B, E, T \vdash e_1.m \rightsquigarrow (x, b', e), S' \qquad S', B, E, T \vdash e_2 \Downarrow v, S'' \\ B \ni \{b \mapsto \rho\} \\ S'' \oplus \{s \mapsto v\}, \{b' \mapsto \rho\}, \{x \mapsto s\}, \{\texttt{return} \mapsto t\} \vdash e \text{ handle } t \Downarrow r, S''' \\ (s, t \text{ fresh}) \end{array}}{S, B, E, T \vdash e_1.m(e_2, \texttt{\&}b) \Downarrow r, S'''}$$

$$\textsc{seq}\quad \frac{S, B, E, T \vdash e_1 \Downarrow v, S' \qquad S', B, E, T \vdash e_2 \Downarrow r, S''}{S, B, E, T \vdash e_1;\ e_2 \Downarrow r, S''}$$

$$\textsc{jump}\quad \frac{S, B, E, T \vdash e \Downarrow v, S' \qquad T \ni \{j \mapsto t\}}{S, B, E, T \vdash j\ e \Downarrow [v]^t, S'}$$

$$\textsc{handle}\quad \frac{S, B, E, T \vdash e \Downarrow v, S'}{S, B, E, T \vdash e \text{ handle } t \Downarrow v, S'} \qquad \textsc{catch}\quad \frac{S, B, E, T \vdash e \Downarrow [v]^t, S'}{S, B, E, T \vdash e \text{ handle } t \Downarrow v, S'}$$

Fig. 2. Evaluation rules of the control calculus

oracle relation $\rightsquigarrow$ is replaced with the counterpart provided by other calculus. The resulting CALL rule of this construction is as follows:

$$\frac{\begin{array}{r} H, B, E, T \vdash e_1 \Downarrow v_1, H' \qquad H' \ni \{v_1 \mapsto (\mathcal{I}, k)\} \\ \mathit{methods}(H', k) \ni \{m \mapsto (x, b', e)\} \qquad H', B, E, T \vdash e_2 \Downarrow v_2, H'' \qquad B \ni \{b \mapsto \rho\} \\ H'' \oplus \{s \mapsto v_2\}, \{b' \mapsto \rho\}, \{x \mapsto s\}, \{\texttt{return} \mapsto t\} \vdash e \text{ handle } t \Downarrow r, H''' \\ (s, t \text{ fresh}) \end{array}}{H, B, E, T \vdash e_1.m(e_2, \texttt{\&}b) \Downarrow r, H'''}$$

The complete definition of the core Ruby calculus is given in Appendix.

4 Extension to the core calculi

To obtain the full Ruby calculus that embodies all of the Ruby's sophisticated language features, this section extends the core calculi with most of Ruby's features. Due to space limitations, this section presents brief summary of each extension. The definitions of the extended calculi are given in Appendix.

Modules and mix-ins. A *module* in Ruby is a structure consisting of a mutable set of methods to be included in a class. By including modules in a class, Ruby supports full-fledged mix-in without any additional machinery other than dynamic method look-up mechanism.

We introduce the following terms to the object calculus: `new_module`, which creates a new module with empty set of methods, and `append_feature` e_1 `to` e_2 which mixes module e_1 in class (or module) e_2. We use the `def` e`#`m `=` ξ syntax to define method m in a module e as well as in a class.

We define the semantic structure of modules by reusing those of classes. Let d range over a given countably infinite set of *module identifiers*. Let δ range over the union of module identifiers and class identifiers, that shall be used for the common behavior between modules and classes. We refer to the union set as *class-module identifiers*.

In Ruby, every class or module has a mutable list of modules included in the class or module. Unlike class inheritance, the relation on module inclusions is not transitive; all affective included modules of a class or module are flatly listed in the included module list. To represent this list, we introduce an *included module list context*, ranged over by L, as a function from class-module identifiers to lists of module identifiers. As in the class inheritance context, we introduce the well-formedness condition on L as follows: for any $\delta' \in \mathrm{dom}(L)$, no identical δ occurs twice in $[\delta'] +\!\!+ L(\delta')$. This prevents circular and doubled inclusion of modules. We note that mutual inclusion of modules is allowed and it does not arise any circular references since module inclusions are not transitive. The only case causing circular references is the case that a module includes itself. The above well-formedness condition clearly prevents this.

A class context $\mathcal{K}$ is redefined to be a *class-module context* of the form (K, L, M) to represent the included modules in each class. We also redefine the structure of a method binding M to be a function from class-module identifiers. Since modules are mutable values, a value v can be a module identifier d. We can define evaluation rules for the mix-in terms by the above preparation.

Implicit destinations of method definitions. In the actual Ruby, the destination class or module of a `def` is implicitly given through a `class` or `module` surrounding the method definition. However, the destination cannot be made up for by syntactic elaboration since `def` can be nested and combined with some reflection primitives. To see the subtle difficulty, consider the following example.

```
class Foo
  def define_bar(c)
    c.hoge { def bar ; "bar" ; end }   # where to define "bar" method?
  end
end
```

The destination of `def bar` $\cdots$ `end` depends on the run-time context. As an uncommon case, if `c` is a module object and `hoge` is an alias to `Module#class_eval` then `bar` is defined in the module denoted by `c`. Otherwise, it is defined in `Foo`.

To specify this behavior of the `def` in our calculi, we extend the object calculus with the feature to keep track of the current destination of `def` terms. We introduce the following new syntax: `module` e_1; e_2; `end` which specifies an existing class or module e_1 as the current destination of `def`s in e_2, and `current_module` which gets out the current destination as a value. For the semantics of these new constructs, we introduce a new evaluation context, *class-module list*, denoted by C, as a list of class-module identifiers. The evaluation relation becomes $H, C \vdash e \Downarrow r, H'$. To keep a class-module list along with each method entity, we introduce a *method closure*, ranged over by ζ, as a pair (C, ξ). We extend a method binding M to be a function to the method closures. Due to this extension, the CALL rule is modified so that the class-module list in a method closure is resumed when evaluating its method body.

Constants and class-variables. Let c range over the set of *constant identifiers.* Any Ruby construct that refers to a constant can be represented with the following two expressions: one is just c, which looks up a given constant from the current class-module list, and another is e `::` c that looks up c from specified module e. Mutations of constants can be essentially represented as single expression of the form e_1 `::` $c = e_2$ which binds c to value e_2 in module e_1.

We extend the class-module context for constants as follows. Let $\mathcal{J}$ be a *constant bindings* as a finite function from constant identifiers to values. Let J be a *constant heap* as a function from class-module identifiers to constant bindings. Then a class-module context $\mathcal{K}$ is extended to be a tuple (J, K, L, M). To define the rule of constant look-ups, a function is needed to compute available constant bindings. This can be defined similarly to the *methods* function according to the standard constant look-up rules. We can define the terms and rules for class variables by introducing class variable identifiers, contexts and look-up mechanisms similarly to the constants.

Method visibility and `super` method calls. Every method in Ruby has its own visibility, which is one of `public`, `protected` or `private`. The visibility controls method lookup of visibility conscious method invocations. In addition, there are two special forms of method invocations in Ruby: m`()` for self method calls and `super` for calling a method bound in a super class. Any of these method invocations can be characterized by how they search for the method to be invoked. To support all variations of Ruby's method invocations, we add two method call terms of the form `send` $e.m$ for visibility conscious method lookup and `super` m for method lookup from super classes. The evaluation rules of these can be derived from the CALL rule by replacing the computation of available method bindings. To deal with the method visibility, we extend the class-module context $\mathcal{K}$ with a *visibility binding*, that is a map from method identifiers to method visibilities. Visibilities of the available methods can be computed by folding the ancestors list with visibility binding composition. We can obtain a visibility conscious available method binding by filtering out invisible methods from the available method binding.

Singleton classes. A *singleton class* is a mechanism to associate a particular method to an object. In Ruby, a singleton class is a class with several minor limitations. Here we only list some of them: they cannot have any instance objects, they cannot be inherited (the standard says *unspecified*), and the visibility of their `protected` methods are unspecified. Exact conformity to all these details requires the operational semantics to distinguish singleton classes from ordinary classes, and re-define class manipulation rules for singleton classes. By omitting these details for simplicity, the singleton classes can be taken into account by extending the object entity to be a triple $(\mathcal{I}, k, k')$ where k' is its singleton class and adding a term for accessing to this k'. The ALLOC and CALL rules are also modified to be conscious of singleton classes along with these extensions.

Miscellaneous control constructs. Ruby has a variety of control flow constructs other than method calls and blocks such as `if` and `unless` expressions, `while` and `until` loops, and user-level exceptions. It is straightforward to incorporate these constructs to the control calculus by representing their control flows in meta-level exceptions. Here we present a set of primitive constructs sufficient for representing Ruby's control flows defined as follows

$$\begin{aligned} e &::= \cdots \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e\ \texttt{end} \mid n{:}\{\, e \,\} \mid e\ \texttt{rescue}\ x.e \mid e\ \texttt{ensure}\ e \\ j &::= \cdots \mid \texttt{redo}_n \mid \texttt{break}_n \mid \texttt{raise} \end{aligned}$$

where n ranges over the set of natural numbers, each of which is used as a loop label. $n{:}\{\, e \,\}$ indicates a loop labeled with n. $\texttt{redo}_n$ jumps to the beginning of the inner-most loop labeled with n. $\texttt{break}_n$ breaks out of the inner-most n-labeled loop. These primitives corresponds to Ruby's `while` or `until` and their `break`, `next` and `redo` statements. `rescue`, `ensure` and `raise` are for user-level exceptions. The evaluation rules for these constructs are trivial and we omit them. Actual Ruby's control flow constructs are introduced by the syntactic elaboration to these primitives.

5 The Ruby calculus

The extended two calculi combine into a calculus in the same way as the core Ruby calculus. The full Ruby calculus is obtained by adding the following three features to the combined calculus: `Proc` objects, reflections, and error handling.

`Proc` objects. In Ruby, a block is not only a control structure but a first-class value. It is straightforward to incorporate this feature to the Ruby calculus by adding block identifiers b to the set of values. The constructs for blocks are modified so that they generate and consume a block value. This is done by replacing the block identifier b in $\texttt{proc}\ \{\, |x|\ e_1 \,\}\ \texttt{as}\ b\ \texttt{in}\ e_2$ with variable x and those in $\texttt{yield}\ e\ \texttt{to}\ b$ and $e_1.m(e_2, \texttt{\&}b)$ with expression e.

Since blocks are first-class values and block values are represented by block identifiers, a block context B must be included in H. In addition, due to the introduction of class-module list mentioned in Section 4, a block entity ρ is extended to be a tuple (E, T, C, x, e). The evaluation relation of the resulting Ruby calculus is of the form $H, E, T, C \vdash e \Downarrow r, H'$. The evaluation rules of the new `proc` and `yield` constructs are given below.

$$\frac{\begin{array}{l} H \oplus \{b \mapsto (E, T \oplus \{\texttt{break} \mapsto t\}, C, x_1, e_1)\} \oplus \{b \mapsto (\{\}, \texttt{Proc})\} \oplus \{s \mapsto b\}, \\ \quad E \oplus \{x_2 \mapsto s\}, T, C \vdash e_2 \text{ handle } t \Downarrow r, H' \qquad\qquad (s, t \text{ fresh}) \end{array}}{H, E, T, C \vdash \texttt{proc} \ \{\ |x_1|\ e_1\ \} \ \texttt{as} \ x_2 \ \texttt{in} \ e_2 \Downarrow r, H'}$$

$$\frac{\begin{array}{c} H, E, T, C \vdash e_2 \Downarrow b, H' \qquad H' \ni \{b \mapsto (E', T', C', x, e)\} \\ H', E, T, C \vdash e_1 \Downarrow v, H'' \\ H'' \oplus \{s \mapsto v\}, E' \oplus \{x \mapsto s\}, T' \oplus \{\texttt{next} \mapsto t\}, C' \vdash e \text{ handle } t \Downarrow r, H''' \\ \hfill (s, t \text{ fresh}) \end{array}}{H, E, T, C \vdash \texttt{yield} \ e_1 \ \texttt{to} \ e_2 \Downarrow r, H'''}$$

The CALL rule is also modified so that it obtains a block parameter through evaluating an expression. This modification is straightforward and we omit it.

Built-in primitives for reflections. Ruby provides a set of built-in primitives for reflecting and reifying evaluation context. The Ruby calculus can readily support these meta-level primitives uniformly. In most compiled languages, a large part of evaluation contexts are static and are consumed away by the compiler. We observe that this is the major source of difficulties in providing meta-level primitives in static languages. In contrast, the operational semantics of our Ruby calculus retains most of meta-level information in its evaluation context, particularly, in heap H. It is then a simple matter to provide reflection functionalities as primitives to examine and modify the current evaluation context. We have studied the meta-level primitives specified in the standard of Ruby and confirmed that in most cases this is indeed the case. In the following, we list typical Ruby's primitive methods and the required primitive constructs in the Ruby calculus.

1. Primitives to access module and class information, such as `Module#instance_methods` and `Module#private`. Since all the information on modules are in the heap, these methods are easily supported by introducing primitives to access to the class-module context in the heap.
2. Primitives to reify the current run-time environment, such as `Kernel.local_variables` and `Kernel.block_given?`. These methods refer to the evaluation context of the caller. Since they do not modify the context, they can be supported through corresponding primitives in the Ruby calculus.
3. Primitives to manipulate the callers' run-time environment, such as `Module#public` and `String#=~`. To support these methods, we first introduce reserved local variables such as `$~` and introduce primitives for manipulating the reserved variables. We note that the standard requires that regular expression matching method dynamically create a local variable binding named `~` in the callers' context. We would say this is unnatural, and this is neither the case in the widely-accepted Ruby implementations. Although approach mentioned above is not strictly compilant with the standard, it would more naturally describe the intuitive meaning of the reserved variables.
4. Primitives to evaluating program texts such as `Kernel.require` and `Kernel.eval`. These allow to execute arbitrary program codes. However, since the results do not affect the caller except for the modification to the heap, they can be supported similarly to the previous case by introducing new primitives.

These reflection mechanisms can be provided through primitive methods. Since primitive methods are themselves possible targets of reflection, the user code can manipulate them. This situation can be dealt with by materializing primitive methods as semantic objects, and separate the evaluation rule of primitive method calls from that of ordinary method calls.

Error handling. The operational semantics of our Ruby calculus may get stuck, yielding *wrong*. In Ruby, however, program never stuck except for uncaught exception. Erroneous cases are reported to the user code through either user-level exceptions, fallback method calls, or default values. If the conformity to this specification is really desired, then it is straightforwardly achieved by introducing appropriate exception for each erroneous case and making all the cases that yield *wrong* in the Ruby calculus as explicit rules that raise the corresponding exception.

The only subtle case is Ruby's `LocalJumpError` exception. This exception is raised when a non-local jump cannot find appropriate jump destinations. This situation mainly occurs in the case where `break` is evaluated in a `Proc` object detached from the evaluation context. In this case, the operational semantics needs to maintain the set of effective exception tags in the evaluation context.

Corner cases requiring extensions. As we have discussed in this section, our conclusion is that the Ruby calculus so far defined provides sufficient basis for a complete specification of the full real Ruby language. The remaining works towards the complete specification is to define semantic objects and evaluation rules for each of the minor details and corner cases that we does not cover in this paper. We list below some of those we found important during our close scrutiny of the standard of Ruby.

- To deal with `Symbol` objects, identifiers should be treated as values.
- `undef` can be dealt with a new "undefined" state in method closures.
- Global variables can be added by introducing a new environment.
- `self` can be treated as a local variable implicitly appearing in the parameter list of a method definition. To pass `self` along with user-specified arguments, methods should be extended to multiple arguments.
- Multiple assignments and multiple arguments can be dealt with by incorporating `Array` objects and their primitives to the semantics.

6 Elaborating Ruby to the Ruby calculus

Ruby allows a number of shorthands and implicit references to various entities, resulting in syntactic ambiguities. The Ruby standard specifies how the ambiguities be resolved during the interpretation of the (abstract) syntax tree. A systematic way to incorporate this ambiguity resolution is the introduction of syntactic elaboration, which translates the real Ruby syntax to terms in the Ruby calculus. We have designed the elaboration algorithm for major Ruby constructs as a series of functions of the form $[\![M]\!]^{\kappa}_{\mathcal{E}}$ that translates Ruby program M into a Ruby calculus term under given elaboration context $\mathcal{E}$, where κ is a name of an elaboration function. The structure of an elaboration context $\mathcal{E}$ is specific to each κ. In this paper, we pick up below some highlights of the elaboration algorithm that demonstrates how it resolves the Ruby's syntactic ambiguities.

Local variable scopes. In Ruby, variable identifiers in reference position are either local variable references or private method calls with no argument. Assignment syntax such as `x =` M implicitly declares that variable `x` is bound in certain (possibly nested) scope. This syntactic ambiguity is resolved by Ruby's scoping rules. Since the scoping rules are all syntactic, this resolution process can be factored into the elaboration algorithm.

We resolve this ambiguity in two steps. At first, we resolve the ambiguity of identifiers by adding explicit empty argument lists to private method calls. This step is defined as a context-sensitive Ruby-to-Ruby translation $[\![M]\!]^{\mathsf{pre}}_V$, where V is the set of identifiers that are decided as variable references. We show a few cases of $[\![M]\!]^{\mathsf{pre}}_V$ below

$$[\![x]\!]^{\mathsf{pre}}_V = \begin{cases} x & \text{if } x \in V \\ x() & \text{if } x \notin V \end{cases} \qquad [\![m\ \texttt{\{ |}x\texttt{|}\ M\ \texttt{\}}]\!]^{\mathsf{pre}}_V = m\ \texttt{\{ |}x\texttt{|}\ [\![M]\!]^{\mathsf{pre}}_{\{x\}}\ \texttt{\}}$$

$$[\![\texttt{for}\ x\ \texttt{in}\ M_1\ \texttt{do}\ M_2\ \texttt{end}]\!]^{\mathsf{pre}}_V = \texttt{for}\ x\ \texttt{in}\ [\![M_1]\!]^{\mathsf{pre}}_{V\cup\{x\}}\ \texttt{do}\ [\![M_2]\!]^{\mathsf{pre}}_{V\cup\{x\}\cup\mathrm{BV}(M_1)}\ \texttt{end}$$

where BV(M) is the set of bound variables occurring in M, some of whose cases are defined as follows.

$$\mathrm{BV}(x\ \texttt{=}\ M) = \{x\} \cup \mathrm{BV}(M) \qquad \mathrm{BV}(m\ \texttt{\{ |}x\texttt{|}\ M\ \texttt{\}}) = \emptyset$$
$$\mathrm{BV}(\texttt{for}\ x\ \texttt{in}\ M_1\ \texttt{do}\ M_2\ \texttt{end}) = \{x\} \cup \mathrm{BV}(M_1) \cup \mathrm{BV}(M_2)$$

The second step is the function $[\![M]\!]^{\mathsf{top}}_V$ that decides the set V of bound variables for each local variable scope and insert `bind` of the Ruby calculus to make the scope explicit. Some cases of this algorithm are as follows

$$[\![m\ \texttt{\{ |}x\texttt{|}\ M\ \texttt{\}}]\!]^{\mathsf{top}}_V = m\ \{\,|x|\ \mathit{bind}\ \mathrm{BV}(M) \setminus (\{x\} \cup V)\ \mathit{in}\ [\![M]\!]^{\mathsf{block}}_{V\cup\mathrm{BV}(M)\cup\{x\}}\,\}$$
$$[\![\texttt{for}\ x\ \texttt{in}\ M_1\ \texttt{do}\ M_2\ \texttt{end}]\!]^{\mathsf{top}}_V =$$
$$\qquad [\![M_1]\!]^{\mathsf{top}}_V\texttt{.each}\ \{\,|x'|\ (\texttt{update}\ x\ \texttt{=}\ x';\ [\![M_2]\!]^{\mathsf{block}}_V)\,\} \quad (x' \text{ fresh})$$

where *bind* V *in* $\cdots$ is the sequence of `bind` x `= nil in` $\cdots$ for all x in V.

Overloaded constructs. Ruby contains many overloaded constructs so that the user can enjoy productive programming with less keywords. Some of the overloaded constructs are resolved by their syntactic context. The elaboration phase is adequate to perform this kind of resolution. Typical examples include `break` and `next` which break out of either `while` loops or blocks. The resolution is expressed as follows.

$$[\![\texttt{while}\ e_1\ \texttt{do}\ e_2\ \texttt{end}]\!]^{\mathsf{top}}_{\mathcal{E}} = 1{:}\{\ \texttt{if}\ [\![e_1]\!]^{\mathsf{while}}_{\mathcal{E}}\ \texttt{then}\ ([\![e_2]\!]^{\mathsf{while}}_{\mathcal{E}};\ \texttt{redo}_1)\ \texttt{else nil}\ \}$$
$$[\![\texttt{break}]\!]^{\mathsf{while}}_{\mathcal{E}} = \texttt{break}_1\ \texttt{nil} \qquad [\![\texttt{break}]\!]^{\mathsf{block}}_{\mathcal{E}} = \texttt{break nil}$$
$$[\![\texttt{next}]\!]^{\mathsf{while}}_{\mathcal{E}} = \texttt{redo}_1\ \texttt{nil} \qquad [\![\texttt{next}]\!]^{\mathsf{block}}_{\mathcal{E}} = \texttt{next nil}$$

For other overloaded constructs, their behavior is selected at run-time. A typical example of this kind is `class` statement in Ruby, which creates a new class or denotes an existing class. There are two design choices to represent this kind of resolution; one is to introduce a new primitive, and the other is to elaborate the `class` statement to a combination of primitives. We choose the latter since this yields a simpler calculus. We show the elaboration rule of `class` statement below

```
[[class $c$ < $e_1$; $e_2$; end]]$^{\mathsf{top}}_{\mathcal{E}}$ =
  bind $x_1$ = [[$e_1$]]$^{\mathsf{top}}_{\mathcal{E}}$ in
  bind $x$ = if defined_const?($c$)
           then if eq(superclass_of($c$), $x_1$) then $c$
                                                else raise TypeError
           else current_module :: $c$ = new_class $x_1$
  in module $x$; [[$e_2$]]$^{\mathsf{top}}_{\mathcal{E}}$; end                     ($x_1, x$ fresh)
```

where `defined_const?`(c), `superclass_of`(e) and `eq`(e_1, e_2) are built-in reify primitives. `defined_const?`(c) returns true if look-up of constant c is succeeded in current context. `superclass_of`(e) returns the direct super class of class e. `eq`(e_1, e_2) returns true if two object e_1 and e_2 are identical.

7 Conformity evaluation

In order to evaluate conformity of our formalism to the actual Ruby language, we developed a prototype Ruby interpreter and carried out an experiment.

Our interpreter is written in Standard ML. Since exceptions in Standard ML are generative, our implementation is a straightforward coding of the evaluation relations so far defined as a recursive evaluation function. We have implemented all of the core Ruby calculus presented in Section 3.3, the large part of extensions presented in Section 4 and 5, and the syntactic elaboration phase described in Section 6. Our prototype interpreter also includes several built-in classes and methods including `Array` and `String`.

We collected test cases from the test case suite distributed with CRuby (a.k.a., MRI, Matz Ruby Implementation) [15] version 1.9.3. CRuby is a de facto standard implementation of Ruby. We selected this test suite on considering that the JIS/ISO standard is largely based on CRuby version 1.8 and 1.9. This test suite consists of 2,729 test cases for both language features and built-in class libraries. We picked up all the test cases for behaviors of blocks and jumps as follows. Firstly, we excluded test cases essentially not for blocks but for methods that use blocks. Secondly, some of test cases use functionalities that our implementation does not support; hence, we modified them so as to conform to our implementation without changing their objectives. For example, we modify methods that take more than one arguments to those that take an array of arguments. Lastly, we had to omit a few test cases, such as those that heavily use variable-length arguments and hashes. This selection and modification gave us 28 test cases.

Our implementation passed 26 out of 28 test cases. The two failures are caused because of the following reasons. One is the case of `LocalJumpError` we discussed in Section 5. This is an expected case; this case can be treated by extending the operational semantics according to the strategy described in Section 5. The other one is a block that takes a block as its argument as follows.

```
Proc.new { |&b| b.call(10) }.call { |x| x }
```

Actually, this program does not follow the JIS/ISO standard, while this feature is available in CRuby from version 1.8.7. Our implementation can support it by a multiple-argument extension, which can certainly be incorporated.

We have seen that the behavior of our implementation is nearly the same as an actual Ruby implementation, and moreover, the exceptional differences can be fixed by small modifications. This result shows promise of our approach.

8 Related works

We owe much of this work to the JIS/ISO standard of Ruby [10, 8] written by the designers and developers of the language. As we mentioned, although it is not a formal specification and it is presented as a description of an abstract machine in a particular implementation style in mind, it provides detailed description of the language. Apart from this document, a few works have been done toward rigorous specification of Ruby. James and Sabry [9] proposed a continuation-based encoding of a generalized `yield` operator similar to the one found in Ruby. This account provides an interesting insight into general nature of iterators with `yield` operator. As we have shown in our development, Ruby's control structure including `yield` is directly represented by generative exceptions rather than continuations, and this scales to various other features of Ruby. Furr et. al. and An et. al. investigated static typing and type inference for a subset of Ruby [4, 6, 2, 3] using their Ruby program analysis framework [5]. These works assume a simplified calculus with the features similar to Ruby. We expect that a formal operational semantics we have worked out in this paper should be beneficial for extending these works on the full set of Ruby.

Our work can be regarded as an attempt to develop a formal semantics of dynamic languages. In this general perspective, a number of works have been done on dynamic scripting languages, including JavaScript such as [12, 7], and Python such as [14], to mention a few. Our approach of modeling a dynamic scripting language as a combination of multiple calculi would shed some light on understanding semantic structures of scripting languages.

9 Conclusions

We have developed a formal operational semantics for Ruby based on the observation that Ruby's dynamic behaviors can be cleanly represented by the composition of two calculi: the object calculus that represent dynamic structures of objects, and the control calculus that accounts for Ruby's control operators including `yield`s. By constructing each of the two calculi in such a way that the features external to the calculus are represented as oracle primitives, two calculi combine trivially to yield the Ruby calculus that represents the essence of

Ruby. This combined calculus scales up to the full Ruby specified in the JIS/ISO standard. The construction of the calculus straightforwardly leads to an implementation in a functional language with generative exceptions. Our evaluation using the implementation indicates that the presented semantics with respect to blocks and `yield`s conforms to commonly accepted Ruby behavior. we plan to make a complete formal specification of Ruby according to the strategy described in Section 5 and present it elsewhere.

An interesting future work is to develop a general framework for specifying a complex language as a composition of multiple orthogonal calculi. Since many existing dynamic languages tend to contain various features, the approach shown in this paper would shade some light toward this direction.

References

1. Abadi, M., Cardelli, L.: *A Theory of Objects.* Springer, Heidelberg (1996)
2. An, J.-H., Chaudhuri, A., Foster, J. S.: Static typing for Ruby on Rails. In: IEEE/ACM International Conference on Automated Software Engineering, pp. 590–594 (2009)
3. An, J.-H., Chaudhuri, A., Foster, J. S., Hicks, M.: Dynamic inference of static types for Ruby. In: ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages, pp. 459–472 (2011)
4. Furr, M., An, J.-H., Foster, J. S.: Profile-guided static typing for dynamic scripting languages. In: ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications, pp. 283–300 (2009)
5. Furr, M., An, J.-H., Foster, J. S., Hicks, M.: The Ruby intermediate language. In: Symposium on Dynamic Languages, pp. 89–98 (2009)
6. Furr, M., An, J.-H., Foster, J. S., Hicks, M.: Static type inference for Ruby. In: ACM symposium on Applied Computing, pp. 1859–1866 (2009)
7. Guha, A., Saftoiu, C., Krishnamurthi, S.: The essence of JavaScript. In D'Hondt, T. (ed.) ECOOP'10. LNCS, vol. 6183, pp. 126–150. Springer, Heidelberg (2010)
8. ISO/IEC 30170:2012, Information technology – Programming languages – Ruby (2012)
9. James, R. P., Sabry, A.: Yield: Mainstream delimited continuations. In: Workshop on the Theory and Practice of Delimited Continuations, pp. 20–32 (2011)
10. JIS X 3017:2011, Programming languages – Ruby (2011)
11. Kahn, G.: Natural semantics. In: Symposium on Theoretical Aspects of Computer Science, pp. 22–39 (1987)
12. Maffeis, S., Mitchell, J. C., Taly, A.: An operational semantics for JavaScript. In: Ramalingam, G. (ed.) APLAS'08. LNCS, vol. 5356, pp. 307–325. Springer, Heidelberg (2008)
13. Milner, R., Tofte, M., MacQueen, D.: *The Definition of Standard ML.* MIT Press (1997)
14. Politz, J. G., Martinez, A., Milano, M., Warren, S., Patterson, D., Li, J., Chitipothu, A., Krishnamurthi, S.: Python: the full monty. In: ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications, pp. 217–232 (2013)
15. Ruby programming language. `http://www.ruby-lang.org/en/`

Appendix A: The complete definition of the core Ruby calculus

The set of terms

$$\begin{array}{rcl}
k & \in & \text{ClassId} \\
m & \in & \text{MethodId} \\
i & \in & \text{InstVarId} \\
x & \in & \text{VarId} \\
b & \in & \text{BlockId} \\
e & \in & \text{Exp} \\
j & \in & \text{JumpKind} \\
\xi \text{ or } (x, b, e) & \in & \text{Method} = \text{VarId} \times \text{BlockId} \times \text{Exp}
\end{array}$$

$$\begin{array}{rl}
e ::= & k \mid e.i \mid e.i \texttt{ = } e \mid \texttt{alloc } e \mid \texttt{new\_class } e \mid \texttt{def } e\texttt{\#}m \texttt{ = } \xi \\
& \mid \texttt{bind } x \texttt{ = } e \texttt{ in } e \mid x \mid \texttt{update } x \texttt{ = } e \\
& \mid \texttt{proc } \{\, |x|\ e \,\} \texttt{ as } b \texttt{ in } e \mid \texttt{yield } e \texttt{ to } b \mid e.m(e, \texttt{\&}b) \\
& \mid e;\ e \mid j\ e \\
j ::= & \texttt{return} \mid \texttt{break} \mid \texttt{next}
\end{array}$$

Semantic objects

$$\begin{array}{rcl}
\mathcal{M} & \in & \text{MethodEnv} = \text{MethodId} \to \text{Method} \\
M & \in & \text{ClassId} \to \text{MethodEnv} \\
K & \in & \text{ClassId} \to \text{ClassId} \\
\mathcal{K} & ::= & (K, M) \\
a & \in & \text{ObjectId} \\
v & \in & \text{Value} = \text{ObjectId} \cup \text{ClassId} \\
\mathcal{I} & \in & \text{InstVarEnv} = \text{InstVarId} \to \text{Value} \\
(\mathcal{I}, k) & \in & \text{Object} = \text{InstVarEnv} \times \text{ClassId} \\
I & \in & \text{Value} \to \text{Object} \\
s & \in & \text{VarRefId} \\
S & \in & \text{VarRefId} \to \text{Value} \\
E & \in & \text{VarEnv} = \text{VarId} \to \text{VarRefId} \\
T & \in & \text{TagEnv} = \text{JumpKind} \to \text{ExnTag} \\
\rho & \in & \text{Block} = \text{VarEnv} \times \text{TagEnv} \times \text{VarId} \times \text{Exp} \\
B & \in & \text{BlockId} \to \text{Block} \\
H & ::= & (I, \mathcal{K}, S) \\
t & \in & \text{ExnTag} \\
r & ::= & v \mid [v]^t \mid \textit{wrong}
\end{array}$$

Special semantic objects

- `Class`, `Module` and `Proc` are built-in class identifiers.
- *Special* is the set of built-in class identifiers; $\mathit{Special} = \{\texttt{Class}, \texttt{Module}, \texttt{Proc}\}$

Well-formedness condition on K

K is well-formed if the transitive closure of K is irreflexive.

Functions for method look-up

$$\mathit{ancestors}(H,k) = \begin{cases} \epsilon & \text{if } k \notin \mathrm{dom}(K) \\ \mathit{ancestors}(K,k') +\!\!+ [k] & \text{if } K \ni \{k \mapsto k'\} \end{cases}$$
$$\mathit{methods}(H,k) = \bigoplus \big[M(k') \bigm| k' \in \mathit{ancestors}(K,k) \big]$$

where $(I,(K,M),S) = H$.

Evaluation relations

$$H,B,E,T \vdash e \Downarrow r, H'$$

or

$$H,B,E,T \vdash e \text{ handle } t \Downarrow r, H'$$

The latter optionally captures the exception having tag t.

Rules of the core object calculus

$$\textsc{class}\quad \frac{k \in \mathrm{dom}(K) \qquad k \in \mathrm{dom}(M)}{H,B,E,T \vdash k \Downarrow k, H}$$

$$\textsc{ivar}\quad \frac{H,B,E,T \vdash e \Downarrow v, H' \qquad H' \ni \{v \mapsto (\mathcal{I}, k)\} \qquad \mathcal{I} \ni \{i \mapsto v'\}}{H,B,E,T \vdash e.i \Downarrow v', H'}$$

$$\textsc{istore}\quad \frac{\begin{array}{c} H,B,E,T \vdash e_1 \Downarrow v_1, H' \qquad H',B,E,T \vdash e_2 \Downarrow v_2, H'' \\ H'' \ni \{v_1 \mapsto (\mathcal{I}, k)\} \end{array}}{H,B,E,T \vdash e_1.i \texttt{ = } e_2 \Downarrow v_2, H'' \oplus \{v_1 \mapsto (\mathcal{I} \oplus \{i \mapsto v_2\}, k)\}}$$

$$\textsc{alloc}\quad \frac{H,B,E,T \vdash e \Downarrow k, H' \qquad k \notin \mathit{Special} \qquad a \text{ fresh}}{H,B,E,T \vdash \texttt{alloc } e \Downarrow a, H' \oplus \{a \mapsto (\{\}, k)\}}$$

$$\textsc{newClass}\quad \frac{H,B,E,T \vdash e \Downarrow k', H' \qquad k \text{ fresh}}{\begin{array}{r} H,B,E,T \vdash \texttt{new\_class } e \Downarrow k, H' \oplus \{k \mapsto (\{\}, \texttt{Class})\} \\ \oplus \{k \mapsto k'\} \oplus \{k \mapsto \{\}\} \end{array}}$$

$$\textsc{def}\quad \frac{H,B,E,T \vdash e \Downarrow k, H' \qquad H' \ni \{k \mapsto \mathcal{M}\}}{H,B,E,T \vdash \texttt{def } e\texttt{\#}m \texttt{ = } \xi \Downarrow k, H' \oplus \{k \mapsto \mathcal{M} \oplus \{m \mapsto \xi\}\}}$$

Rules of the core control calculus

$$\textsc{bind}\quad \frac{\begin{array}{c}H, B, E, T \vdash e_1 \Downarrow v, H' \\ H' \oplus \{s \mapsto v\}, B, E \oplus \{x \mapsto s\}, T \vdash e_2 \Downarrow r, H'' \ (s \text{ fresh})\end{array}}{H, B, E, T \vdash \texttt{bind}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \Downarrow r, H''}$$

$$\textsc{var}\quad \frac{E \ni \{x \mapsto s\} \qquad H \ni \{s \mapsto v\}}{H, B, E, T \vdash x \Downarrow v, H}$$

$$\textsc{update}\quad \frac{E \ni \{x \mapsto s\} \qquad H, B, E, T \vdash e \Downarrow v, H'}{H, B, E, T \vdash \texttt{update}\ x\ \texttt{=}\ e \Downarrow v, H' \oplus \{s \mapsto v\}}$$

$$\textsc{proc}\quad \frac{\begin{array}{r}H, B \oplus \{b \mapsto (E, T \oplus \{\texttt{break} \mapsto t\}, x, e_1)\}, E, T \vdash e_2 \text{ handle } t \Downarrow r, H' \\ (t \text{ fresh})\end{array}}{H, B, E, T \vdash \texttt{proc}\ \{\ |x|\ e_1\ \}\ \texttt{as}\ b\ \texttt{in}\ e_2 \Downarrow r, H'}$$

$$\textsc{yield}\quad \frac{\begin{array}{r}H, B, E, T \vdash e \Downarrow v, H' \qquad B \ni \{b \mapsto (E', T', x, e)\} \\ H' \oplus \{s \mapsto v\}, \emptyset, E' \oplus \{x \mapsto s\}, T' \oplus \{\texttt{next} \mapsto t\} \vdash e \text{ handle } t \Downarrow r, H'' \\ (s, t \text{ fresh})\end{array}}{H, B, E, T \vdash \texttt{yield}\ e\ \texttt{to}\ b \Downarrow r, H''}$$

$$\textsc{call}\quad \frac{\begin{array}{r}H, B, E, T \vdash e_1 \Downarrow v_1, H' \qquad H' \ni \{v_1 \mapsto (\mathcal{I}, k)\}^I \\ \mathit{methods}(H, k) \ni \{m \mapsto (x, b', e)\} \\ H', B, E, T \vdash e_2 \Downarrow v_2, H'' \qquad B \ni \{b \mapsto \rho\} \\ H'' \oplus \{s \mapsto v_2\}, \{b' \mapsto \rho\}, \{x \mapsto s\}, \{\texttt{return} \mapsto t\} \vdash e \text{ handle } t \Downarrow r, H''' \\ (s, t \text{ fresh})\end{array}}{H, B, E, T \vdash e_1.m(e_2, \texttt{\&}b) \Downarrow r, H'''}$$

$$\textsc{seq}\quad \frac{H, B, E, T \vdash e_1 \Downarrow v, H' \qquad H', B, E, T \vdash e_2 \Downarrow r, H''}{H, B, E, T \vdash e_1;\ e_2 \Downarrow r, H''}$$

$$\textsc{jump}\quad \frac{H, B, E, T \vdash e \Downarrow v, H' \qquad T \ni \{j \mapsto t\}}{H, B, E, T \vdash j\ e \Downarrow [v]^t, H'}$$

Rules for exception handling

$$\textsc{handle}\quad \frac{H, B, E, T \vdash e \Downarrow v, H'}{H, B, E, T \vdash e \text{ handle } t \Downarrow v, H'}$$

$$\textsc{catch}\quad \frac{H, B, E, T \vdash e \Downarrow [v]^t, H'}{H, B, E, T \vdash e \text{ handle } t \Downarrow v, H'}$$

Appendix B: The definition of the extended object calculus

The set of terms

$c \in \text{ConstId}$

$e ::= k \mid e.i \mid e.i\ \texttt{=}\ e \mid e.m \mid \texttt{alloc}\ e \mid \texttt{new\_class}\ e \mid \texttt{def}\ e\texttt{\#}m\ \texttt{=}\ \xi$
$\quad \mid \texttt{new\_module} \mid \texttt{append\_feature}\ e\ \texttt{to}\ e$
$\quad \mid \texttt{module}\ e;\ e;\ \texttt{end} \mid \texttt{current\_module}$

$$\begin{aligned} &\mid c \mid e::c \mid e::c = e \\ &\mid \texttt{send}\ e.m \mid \texttt{super}\ m \mid \texttt{singleton\_class\_of}\ e \end{aligned}$$

Semantic objects

$$\begin{aligned}
\mathcal{M} &\in \text{MethodEnv} = \text{MethodId} \to \text{MethodClosure} \\
M &\in (\text{ClassId} \cup \text{ModuleId}) \to \text{MethodEnv} \\
L &\in (\text{ClassId} \cup \text{ModuleId}) \to \text{ModuleId}^* \\
\mathcal{J} &\in \text{ConstEnv} = \text{ConstId} \to \text{Value} \\
J &\in (\text{ClassId} \cup \text{ModuleId}) \to \text{ConstEnv} \\
\mathcal{V} &\in \text{VisibilityEnv} = \text{MethodId} \to \text{Visibility} \\
V &\in (\text{ClassId} \cup \text{ModuleId}) \to \text{VisibilityEnv} \\
\mathcal{K} &::= (J, K, L, M, V) \\
C &::= (\text{ClassId} \cup \text{ModuleId})^* \\
\zeta &::= \text{MethodClosure} = (\text{ClassId} \cup \text{ModuleId})^* \times \text{Method} \\
d &\in \text{ModuleId} \\
\delta &\in \text{ClassId} \cup \text{ModuleId} \\
\nu &::= \text{Visibility} = \{\texttt{private}, \texttt{protected}, \texttt{public}\} \\
v &\in \text{Value} = \text{ObjectId} \cup \text{ClassId} \cup \text{ModuleId} \\
(\mathcal{I}, k, k') &\in \text{Object} = \text{InstVarEnv} \times \text{ClassId} \times \text{ClassId}
\end{aligned}$$

Well-formedness condition on L

For any $\delta' \in \text{dom}(L)$, no identical δ occurs twice in $\delta' +\!\!+ L(\delta')$.

Functions for mix-in conscious method look-up

$$\begin{aligned}
\textit{ancestors}(H,k) &= \begin{cases} \epsilon & \text{if } k \notin \text{dom}(K) \\ \textit{ancestors}(H,k') +\!\!+ L(k) +\!\!+ [k] & \text{if } K \ni \{k \mapsto k'\} \end{cases} \\
\textit{methods}(H,k) &= \bigoplus \big[M(\delta) \bigm| \delta \in \textit{ancestors}(H,k) \big]
\end{aligned}$$

Functions for visibility concious method look-up

$$\begin{aligned}
\textit{visibility}(H,k) &= \bigoplus \big[V(\delta) \bigm| \delta \in \textit{ancestors}(H,k) \big] \\
\textit{belong}(H,k) &= \bigoplus \big[V'(\delta) \bigm| \delta \in \textit{ancestors}(H,k) \big] \\
&\qquad \text{where } V'(\delta) = \{ m \mapsto \delta \mid m \in \text{dom}(V(\delta)) \}. \\
\textit{visible}(H,k,m,\delta) &= (\textit{visibility}(H,k) = \texttt{public}) \\
&\quad \vee \big(\textit{visibility}(H,k) = \texttt{protected} \\
&\qquad \wedge\ \textit{belong}(H,k)(m) \in \textit{ancestors}(H,\delta) \big) \\
\textit{methods}'(H,k,\delta) &= \big\{ m \mapsto \zeta \bigm| \textit{methods}(H,k) \ni \{m \mapsto \zeta\}, \textit{visible}(H,k,m,\delta) \big\}
\end{aligned}$$

Functions for super method look-up

$$\mathit{methods}''(H,\delta) = \bigoplus\big[M(\delta') \;\big|\; \delta' \in (\mathit{ancestors}'(H,\delta) \downarrow \delta)\big]$$

where $l \downarrow x = [y \mid y \in l, x \neq y]$.

Functions for constant look-up

$$\begin{aligned} \mathit{csp}(H,k) &= \mathop{+\!\!\!+}\big[L(\delta') \mathbin{+\!\!+} [\delta'] \;\big|\; \delta' \in \mathit{ancestors}(H,k)\big] \\ \mathit{csp}(H,d) &= [\texttt{Object}] \mathbin{+\!\!+} L(d) \mathbin{+\!\!+} [d] \end{aligned}$$

Functions on lists

- $\mathit{nub}(l)$ is the list obtained from l by eliminating duplicate elements and keeping the first occurrence of each element.

Evaluation relation

$$H, C \vdash e \Downarrow r, H'$$

Oracle relation of external evaluation

$$H, C \vdash \xi \rightsquigarrow r, H'$$

Rules of the core object calculus

$$\textsc{class}\quad \frac{k \in \mathrm{dom}(K) \qquad k \in \mathrm{dom}(M)}{H, C \vdash k \Downarrow k, H}$$

$$\textsc{ivar}\quad \frac{H, C \vdash e \Downarrow v, H' \qquad H' \ni \{v \mapsto (\mathcal{I}, k, k')\} \qquad \mathcal{I} \ni \{i \mapsto v'\}}{H, C \vdash e.i \Downarrow v', H'}$$

$$\textsc{istore}\quad \frac{H, C \vdash e_1 \Downarrow v_1, H' \qquad H', C \vdash e_2 \Downarrow v_2, H'' \qquad H'' \ni \{v_1 \mapsto (\mathcal{I}, k, k')\}}{H, B, E, T \vdash e_1.i \texttt{ = } e_2 \Downarrow v_2, H'' \oplus \{v_1 \mapsto (\mathcal{I} \oplus \{i \mapsto v_2\}, k, k')\}}$$

$$\textsc{alloc}\quad \frac{\begin{array}{c} H, C \vdash e \Downarrow k, H' \qquad k \notin \mathit{Special} \\ H'C \vdash \texttt{new\_class k} \Downarrow k', H'' \qquad a \text{ fresh} \end{array}}{H, C \vdash \texttt{alloc } e \Downarrow a, H'' \oplus \{a \mapsto (\{\}, k, k')\}}$$

$$\textsc{newClass}\quad \frac{H, C \vdash e \Downarrow k', H' \qquad k \text{ fresh}}{\begin{array}{r} H, C \vdash \texttt{new\_class } e \Downarrow k, H' \oplus \{k \mapsto (\{\}, \texttt{Class})\} \oplus \{k \mapsto k'\} \\ \oplus \{k \mapsto \{\}\} \oplus \{k \mapsto \{\}\} \oplus \{k \mapsto \{\}\} \\ \oplus \{k \mapsto []\} \end{array}}$$

$$\textsc{def}\quad \frac{H, C \vdash e \Downarrow \delta, H' \qquad H' \ni \{\delta \mapsto \mathcal{M}\} \qquad H' \ni \{\delta \mapsto \mathcal{V}\}}{\begin{array}{r} H, C \vdash \texttt{def } e\texttt{\#}m \texttt{ = } \xi \Downarrow \delta, H' \oplus \{\delta \mapsto \mathcal{M} \oplus \{m \mapsto (C, \xi)\}\} \\ \oplus \{\delta \mapsto \mathcal{V} \oplus \{m \mapsto \texttt{public}\}\} \end{array}}$$

$$\textsc{call}\quad \frac{\begin{array}{c} H, C \vdash e \Downarrow v, H' \qquad H' \ni \{v \mapsto (\mathcal{I}, k, k')\} \\ \mathit{methods}(H', k') \ni \{m \mapsto (C', \xi)\} \qquad H', C' \vdash \xi \rightsquigarrow v, H'' \end{array}}{H, C \vdash e.m \Downarrow v, H''}$$

Rules for mix-ins

$$\textsc{newModule}\quad \frac{d \text{ fresh}}{\begin{array}{l} H \vdash \texttt{new\_module} \Downarrow d, H \oplus \{d \mapsto (\{\}, \texttt{Module})\} \\ \qquad\qquad\qquad\qquad \oplus \{d \mapsto \{\}\} \oplus \{d \mapsto \{\}\} \oplus \{d \mapsto \{\}\} \\ \qquad\qquad\qquad\qquad \oplus \{d \mapsto []\} \end{array}}$$

$$\textsc{append}\quad \frac{\begin{array}{c} H \vdash e_1 \Downarrow d, H' \qquad H' \vdash e_2 \Downarrow \delta, H' \qquad H' \ni \{d \mapsto l_1\} \\ H' \ni \{\delta \mapsto l_2\} \qquad \delta \notin l_1 +\!\!+ [d] \end{array}}{H \vdash \texttt{append\_feature}\ e_1\ \texttt{to}\ e_2 \Downarrow \delta, H' \oplus \{\delta \mapsto \mathit{nub}(l_2 +\!\!+ l_1 +\!\!+ d)\}}$$

Rules for method definitions

$$\textsc{module}\quad \frac{H, C \vdash e_1 \Downarrow \delta, H' \qquad H', C +\!\!+ [\delta] \vdash e_2 \Downarrow v, H''}{H, C \vdash \texttt{module}\ e_1\texttt{;}\ e_2\texttt{;}\ \texttt{end} \Downarrow v, H''}$$

$$\textsc{current}\quad H, C +\!\!+ [\delta] \vdash \texttt{current\_module} \Downarrow \delta, H$$

Rules for variations of method invocations

$$\textsc{send}\quad \frac{\begin{array}{c} H, C \vdash e \Downarrow v, H' \qquad H' \ni \{v \mapsto (\mathcal{I}, k, k')\} \\ H, C \vdash \texttt{current\_module} \Downarrow \delta, H' \qquad \mathit{methods}'(H', k', \delta) \ni \{m \mapsto (C', \xi)\} \\ H', C' \vdash \xi \rightsquigarrow v, H'' \end{array}}{H, C \vdash \texttt{send}\ e.m \Downarrow v, H''}$$

$$\textsc{super}\quad \frac{\begin{array}{c} H, C \vdash \texttt{current\_module} \Downarrow \delta, H' \qquad \mathit{methods}''(H', \delta) \ni \{m \mapsto (C', \xi)\} \\ H', C' \vdash \xi \rightsquigarrow v, H'' \end{array}}{H, C \vdash \texttt{super}\ m \Downarrow v, H''}$$

Rules for singleton classes

$$\textsc{singleton}\quad \frac{H, C \vdash e \Downarrow v, H' \qquad H \ni \{(\mathcal{I}, k, k')\}}{H, C \vdash \texttt{singleton\_class\_of}\ e \Downarrow k', H}$$

Appendix C: The definition of the extended control calculus

The set of terms

$$\begin{array}{rl} e ::= & \cdots \\ & \mid\ \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e\ \texttt{end}\ \mid\ n{:}\{\,e\,\}\ \mid\ e\ \texttt{rescue}\ x.e\ \mid\ e\ \texttt{ensure}\ e \\ j ::= & \cdots \\ & \mid\ \texttt{redo}_n\ \mid\ \texttt{break}_n\ \mid\ \texttt{raise} \end{array}$$

Rules for conditional expressions

$$\textsc{if-then}\quad \frac{H, B, E, T \vdash e_1 \Downarrow v, H' \qquad v \notin \mathit{False} \qquad H', B, E, T \vdash e_2 \Downarrow r, H''}{H, B, E, T \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3\ \texttt{end} \Downarrow r, H''}$$

$$\textsc{if-else}\quad \frac{H, B, E, T \vdash e_1 \Downarrow v, H' \qquad v \in \mathit{False} \qquad H', B, E, T \vdash e_3 \Downarrow r, H''}{H, B, E, T \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3\ \texttt{end} \Downarrow r, H''}$$

Rules for iteration expressions

$$\textsc{loop1}\quad \frac{\begin{array}{c} T' = T \oplus \{\texttt{redo}_n \mapsto t_1, \texttt{break}_n \mapsto t_2\}\ (t_1, t_2 \text{ fresh}) \\ H, B, E, T' \vdash e \text{ handle } t_2 \Downarrow v, H' \end{array}}{H, B, E, T \vdash n{:}\{\, e \,\} \Downarrow v, H'}$$

$$\textsc{loop2}\quad \frac{\begin{array}{c} T' = T \oplus \{\texttt{redo}_n \mapsto t_1, \texttt{break}_n \mapsto t_2\}\ (t_1, t_2 \text{ fresh}) \\ H, B, E, T' \vdash e \text{ handle } t_2 \Downarrow [v]^{t_1}, H' \\ H', B, E, T \vdash n{:}\{\, e \,\} \Downarrow r, H'' \end{array}}{H, B, E, T \vdash n{:}\{\, e \,\} \Downarrow r, H''}$$

Rules for user-level exceptions

$$\textsc{rescue}\quad \frac{T' = T \oplus \{\texttt{raise} \mapsto t\}\ (t \text{ fresh}) \qquad H, B, E, T \vdash e_1 \Downarrow v, H'}{H, B, E, T \vdash e_1 \ \texttt{rescue}\ x.e_2 \Downarrow v, S'}$$

$$\textsc{rescue}\quad \frac{\begin{array}{c} T' = T \oplus \{\texttt{raise} \mapsto t\}\ (t \text{ fresh}) \qquad H, B, E, T \vdash e_1 \Downarrow [v]^t, H' \\ S' \oplus \{s \mapsto v\}, E \oplus \{x \mapsto s\}, T \vdash e_1 \Downarrow r, H'' \end{array}}{H, B, E, T \vdash e_1 \ \texttt{rescue}\ x.e_2 \Downarrow r, H''}$$

$$\textsc{ensure}\quad \frac{H, B, E, T \vdash e_1 \Downarrow r_1, H' \qquad H', B, E, T \vdash e_2 \Downarrow r_2, H''}{H, B, E, T \vdash e_1 \ \texttt{ensure}\ e_2 \Downarrow r_1, H''}$$