Ruby Object Model – The S1 structure

Ondřej Pavlata

April 23, 2012

Abstract An algebraic structure is described that forms the innermost core of the object model of the Ruby programming language. The structure, denoted S1, is induced by superclass and eigenclass links between objects. Since Ruby is a serious candidate for the most object-oriented programming language in the world, the S1 structure can be considered to be the *core structure of object technology.*

We provide a simple set-theoretic representation showing how the structure can be interpreted via set inclusion and set membership. We also describe lazy evaluation of infinite eigenclass chains. Finally, the Smalltalk precursor of the S1 structure is discussed.

Keywords Ruby, object model, structure, abstract state machine, monounary algebra, tree, inheritance, superclass, eigenclass, metaclass, Smalltalk-80.

1 Introduction

Mathematical structures, shortly, *structures* [Mul10] are used in science to provide insightful description of phenomena under consideration. Structures are primarily human-centric. To be descriptive they must themselves be described in a way that is comprehensible to a human reader. Most often, complex descriptions can be decomposed into simpler parts. For a structure $\mathcal{S}$, this yields a hierarchy of structures that are *abstractions* of $\mathcal{S}$. The higher in the hierarchy a structure is, the more abstract it is. The more abstract a structure is, the more fundamental it is, because less abstract structures are based on it.

In software engineering, structures naturally arise as a means for description of data. This has been recognized in particular in the theory of Abstract State Machines (ASM) [Gur95] [BS03]. The central point of this theory is a model of program computation in which each state is a mathematical structure. Since structures can be specified incrementally, one can build a chain of increasingly refined models, eventually obtaining a (full) formal specification of the program. This method should serve purposes that fall into 2 main categories:

(A) Human-centric: Provide rigorous implementation-independent description that clarifies the software under consideration.

(B) Machine-centric: Perform automatic program verification (model checking) by processing the formal specification as an executable code.

However, present applications of abstract state machines seem to be predominantly concerned with the (B) case – they are computer-centric. Even case studies like [SSB01] that claim to serve both purposes equally well, seem to be (to the author of this article) significantly affected by runtime orientation. In fact, the author of this article was not able to find a single document (of other authors) that could be considered a relevant human-centric application of structures in the spirit of abstract state machines.

The subject of this article is a structure, denoted S1, that forms the fundamental part of the Ruby object model. The Ruby programming language [FM08] [Bla09] is an extremely sophisticated tool for creating software. It is a complex language with strong support for expressing things as simple and as clear as possible. Being object-oriented, Ruby belongs to the branch of information technology that has abstraction as one of its main concerns. Therefore, the S1 structure can be considered an essence of essence of essence. Hopefully, this article will provide a valid contribution to "structural" foundation of information technology.

1.1 Ruby version

This article refers to Ruby 1.9, more specifically, to the Matz's Ruby Interpreter (MRI) 1.9.2. However, in the S1 structure, there are presumably no differences between 1.9 and previous versions, except for the presence of the `BasicObject` class (so that previous versions contained 3 helix classes instead of 4). Moreover, no changes have been announced for the 2.0 version that would relate to the S1 structure.

2 Preliminaries

2.1 Notational conventions

In most cases we will use Ruby-like "dot" notation for functions (maps). For example, $x.f$ means the value of $.f$ at x. If X is a subset of the domain of $.f$, then $X.f$ denotes the image of X under $.f$, i.e. the set $\{x.f \mid x \in X\}$. This also holds in the case that X happens to also be an element of the domain (the interpretation is indicated by the uppercase notation [1]). We will use left-to-right alignment for relational and functional composition: $R \circ S = \{(r, s) \mid (r, x) \in R$ and $(x, s) \in S$ for some $x\}$. Similarly, $.f.g$ means $(.f) \circ (.g)$.

Symbols $\to$ / $\curvearrowright$ / $\hookrightarrow$ indicate total / partial / injective maps, respectively.

Ruby notation is also used for lists. In particular, $l[i]$ and $l.last$ denote the i-th and the last member of a list l, respectively.

2.2 Monounary algebra

A *(total) monounary algebra* is an algebra with a single unary operation, i.e. it is a structure $(X, .\overline{p})$ such that X is a set and $.\overline{p}$ is a function $X \to X$.

A *partial monounary algebra* is a structure $(X, .p)$ where $.p$ is a *partial* function on X. An element $x \in X$ is called a *fixed point* if $x.p = x$. There is a one-to-one correspondence between total monounary algebras and partial monounary algebras without fixed points: the fixed points become undefined and vice versa. This can be notationally expressed by adding/removing an overline ($.\overline{p} \leftrightarrow .p$).

We denote $x.p(i)$ the i-th application of $.p$ to x (we put $x.p(0) = x$).

[1] An exception to this rule are Ruby constant identifiers, e.g. `B`.*ec* means the eigenclass of `B`.

2.3 Algebraic forest

By an *algebraic forest* (or just *forest*) we mean a structure which has one of the following equivalent forms:

(1) A partial order whose every principal up-set is a finite chain:

A partial order $(X, \leq)$ such that for every $x \in X$, the set $x.ps = \{y \in X \mid x \leq y\}$ is finite and totally ordered by $\leq$.

(2) A monounary algebra whose every non-empty subalgebra has a fixed point:

An algebra $(X, .\overline{p})$ with just one unary operator, $.\overline{p} : X \to X$, such that for every $x \in X$, $x.\overline{p}(i) = x.\overline{p}(i+1)$ for some natural i.

(3) A partial monounary algebra without total non-empty subalgebras:

A partial algebra $(X, .p)$ with just one partial unary operator, the *parent* partial function $.p : X \curvearrowright X$, such that for every $x \in X$, $x.p(i)$ is defined for only finitely many i.

The correspondence (1)←(3) and (1)→(3) is via reflexive transitive closure and reduction, respectively. (2) and (3) correspond via the $.\overline{p} \leftrightarrow .p$ correspondence of (partial) monounary algebras.

Because the set $x.ps$ from (1) is a finite chain in $\leq$, we can regard it as a finite list. The *root* map $.r : X \to X$ is then defined by $x.r = x.ps.last$. Obviously, $.r$ is a closure operator w.r.t. $(X, \leq)$. An element is a *root* if it is $.r$-closed (i.e. $.\overline{p}$-fixed, i.e. $.p$-undefined).

2.4 Algebraic tree

By an *algebraic tree* (or just *tree*) we mean an algebraic forest with exactly one root. As an algebra, it is a structure $(X, .\overline{p}, \underline{r})$, such that $(X, .\overline{p})$ is an algebraic forest and $\underline{r}$ is the only fixed point of $.\overline{p}$.

2.5 Primorder algebra

By a *primorder algebra* we mean a structure $(X, .ec, .pr)$ where X is a set,

$.ec$ is a map $X \to X$, $x.ec$ is called the $\underline{e}igen\underline{c}lass$ of x,
$.pr$ is a map $X \to X$, $x.pr$ is called the $\underline{pr}imary$ *element* of x.

Elements from $X.pr$ (resp. $X.ec$) are called *primary* (resp. *eigenclasses*). The structure is subject to the following axioms:

(1) $.ec$ is injective.

We denote $.ce$ the inverse of $.ec$. If defined, $x.ce$ is the *(direct) eigenclass predecessor* of x.

(2) The partial algebra $(X, .ce)$ is a forest with the root map equal to $.pr$.

We write $x.ec(i)$ for i-th application of $.ec$ to x. The *eigenclass index* of x, denoted $x.eci$, is defined as the depth of x in $(X, .ce)$, i.e. it is the unique i such that $x.pr.ec(i) = x$.

Observations:

(i) Each component of the monounary algebra $(X, .ec)$ is isomorphic (via $.eci$) to the structure $(\mathbb{N}, succ)$ of natural numbers where $succ$ is the successor operator.

(ii) An element is primary iff it is the primary element of itself iff it has no eigenclass predecessors iff it has zero eigenclass index.

(iii) A primorder algebra $(X, .ec, .pr)$ is uniquely given by its reduct $(X, .ec)$.

2.6 State transition system

By a *(labelled) state transition system* we mean structure (C, Act, T, r) where

C is the *state domain* which is a set-theoretic class of (many-sorted) structures with a common *signature*, elements of C are *states*,

Act is the *action domain* which is a set of *actions*,

T is the *transition relation* which is a subset of $C \times Act \times C$,

r is an element of C called the *inital state*.

The *reachability* relation, R^*, is defined as the reflexive transitive closure of the natural projection R of T to $C \times C$. The class of *reachable states* is a subclass D of C that equals the image of the initial state r under R^*.

2.7 Superstructure

For a set X, we denote $\mathbb{P}(X)$ the *powerset* of X, the set of all subsets of X. In addition, we denote

$\mathbb{P}_+(X) \;= \mathbb{P}(X) \setminus \{\emptyset\}$, the set of all non-empty subsets of X,

$\mathbb{P}_\star(X) \;= \mathbb{P}_+(X) \cup X$ (the first quasi-superstructure of X).

We will also use exponents for multiple application, e.g. ${\mathbb{P}_+}^i(X)$ means i-th application of $\mathbb{P}_+$ to X (we put ${\mathbb{P}_+}^0(X) = X$).

We call a set V a *superstructure* [CK90] if it equals the infinite union

$$V \;= \bigcup\{{\mathbb{P}_\star}^i(U) \mid i \in \mathbb{N}\} \quad \text{(so that we could write } V = {\mathbb{P}_\star}^\omega(U))$$

where U is a subset of V such that

- $\emptyset \notin U$,
- $x \cap V = \emptyset$ for every $x \in U$.

This means that U is uniquely given as the set $U(V) = \{x \in V \mid x \cap V = \emptyset\}$. Elements of U can be considered *urelements* (and we will call them so) – their set-theoretic structure does not interfere with the superstructure so that they are atoms in V with respect to set membership.

For $a \in V$, we denote $a.d$ the *rank* (alternatively, *$\underline{d}$epth*) of a to be the smallest i such that $a \in {\mathbb{P}_\star}^i(U)$. Equivalently, $a.d$ is the maximal n such that

$$a_0 \in \cdots \in a_n = a \quad \text{for some } a_0, \ldots, a_n \text{ from } V \text{ (necessarily, } a_0 \text{ is an urelement).}$$

The superstructure is naturaly *stratified* according to the rank function. The set U of urelements is exactly the set of elements with rank 0. For every natural $n > 0$, the difference $\mathbb{P}_\star{}^n(U) \setminus \mathbb{P}_\star{}^{n-1}(U)$ is the set of elements with rank n.

For $a \in V \setminus U$, non-empty intersections of a with strata of V form strata of a. We denote $a.\rho$ the *bottom stratum* of a, i.e.

$$a.\rho = \{x \in a \mid x.d = d\} \text{ where } d = min\{x.d \mid x \in a\}.$$

We also denote $.\overline{\rho}$ the extension of $.\rho$ to V by putting $x.\overline{\rho} = x$ if x is an urelement.

We denote $.ec$ the *eigenclass map* $V \to V$ defined as an "adaptation" of $\mathbb{P}_+$ to urelements as follows:

$$x.ec = \{x\} \text{ if } x \text{ is an urelement,}$$
$$x.ec = \mathbb{P}_+(x) \text{ otherwise.}$$

Proposition

(i) $(V, .ec)$ is (the reduct of) a primorder algebra.

We will apply established notation and terminology, in particular, definition of $.pr$, $.ce$ and $.eci$.

(ii) Urelements and elements containing at least 2 urelements are (among) primary elements.

(iii) For every $x \in V.ec \setminus U.ec$, $\;x.ce = \bigcup x$.

(iv) For every $x, y \in V \setminus U$, if $x \subseteq y$ then $y.\rho.d \le x.\rho.d \le x.d \le y.d$.

(v) For every $x \in V$,

$$x.ec.\overline{\rho} = x.\overline{\rho}.ec,$$

i.e. the eigenclass map commutes with the (extended) bottom stratum map.

(vi) For every $x, y \in V$,

$$x \in y.ec \;\text{ iff }\; x.ec \subseteq y.ec.$$

(vii) For every $x, y \in V \setminus U$, $i \ge 0$,

$$x \subseteq y \;\text{ iff }\; x.ec(i) \subseteq y.ec(i).$$

Recall that we adopt the convention that if a subset X of V is denoted by a capital letter and $.f$ is a function on V then

$X.f = \{x.f \mid x \in X\}$, i.e $X.f$ is the image of X under $.f$ (this might differ from the value of $.f$ at X).

For example, if X is a set of urelements, then $X.ec$ means the set of all singleton subsets of X.

3 The S1 structure

By an *S1 structure* we mean a structure $(\underline{O}, .ec, .pr, .sc, \underline{r}, \underline{c})$ where

- $\underline{O}$ is a set of *objects*.
- $.ec$ is a total function between objects, $x.ec$ is called the $\textit{\underline{e}igen\underline{c}lass}$ of x.
- $.pr$ is a total function between objects, $x.pr$ is called the $\textit{\underline{pr}imary object}$ of x.
- $.sc$ is a partial function between objects, $x.sc$ (if defined) is the $\textit{\underline{s}uper\underline{c}lass}$ of x.
- $\underline{r}$ is an object, called the *inheritance root*.
- $\underline{c}$ is an object, called the *instance root* (by s1(3b), $\underline{c}$ is a shorthand for $\underline{r}.ec.sc$).

Objects from $\underline{O}.pr$ are *primary*. We introduce explicit notation for some distinguished sets of objects.

- $\underline{T}$ is the set of *terminal objects* or just *terminals*. It consists of primary objects x such that $x \neq \underline{r}$ and $x.sc$ is not defined.
- $\underline{C}$ is the set of *classes* – primary objects that are not in $\underline{T}$.
- $\overline{C}$ denotes the set $\underline{C} \cup \underline{T}.ec$.
- $\underline{H}$ is the set of *helix objects* – objects x such that $x.pr = \underline{c}.sc(i)$ for some $i \geq 0$.

The structure is subject to the following conditions:

s1(0) The set $\underline{O}.pr$ of primary objects is finite.

s1(1) $(\underline{O}, .ec, .pr)$ is a primorder algebra.

We will apply established notation and terminology, in particular, definition of $.ce$ and $.eci$.

s1(2) $\underline{r}$ and $\underline{c}$ are different classes.

s1(3) The superclass partial map $.sc$ satisfies the following:

(a) $(\overline{C}, .sc, .\underline{r})$ is an algebraic tree such that

$\underline{r}.sc$ undefined,

elements of $\underline{T}.ec$ are (among the) leaves, i.e. $\underline{T}.ec.sc \subseteq \underline{C} \supseteq \underline{C}.sc$.

(b) $x.sc$ equals $\underline{c}$ if $x = \underline{r}.ec$.

(c) $x.sc$ equals $x.ce.sc.ec$ if $x \in \underline{O}.ec \setminus (\{\underline{r}.ec\} \cup \underline{T}.ec)$.

s1(4) $\underline{c}$ is a leaf of $(\overline{C}, .sc, \underline{r})$, i.e. $\underline{c} \notin \overline{C}.sc$.

Observations:

(i) Condition s1(3c) provides a unique extension of $.sc$ from $X.ec(i)$ to $X.ec(i+1)$, $i \geq 0$, where $X = (\underline{C} \setminus \{\underline{r}\}) \cup (\underline{T}.ec \cup \{\underline{r}.ec\})$.

(ii) Denote $\underline{O}_{01} = \underline{O}.pr \cup \underline{O}.pr.ec$, the *conventional extent* of objects. Up to isomorphism, an S1 structure is uniquely determined by $(\underline{O}_{01}, .ec, .sc)$.

Figure 1 shows a generic S1 structure from the "side view". Each column corresponds to an eigenclass index (so that the display is pruned to primary objects together with 1st to 4th eigenclasses). The lines drawn between rounded boxes represent superclass links. They are directed down-up or right-to-left in the case of $\int$-shaped "twist" links. Eigenclass links are not drawn – they are assumed to go horizontally, left to right.

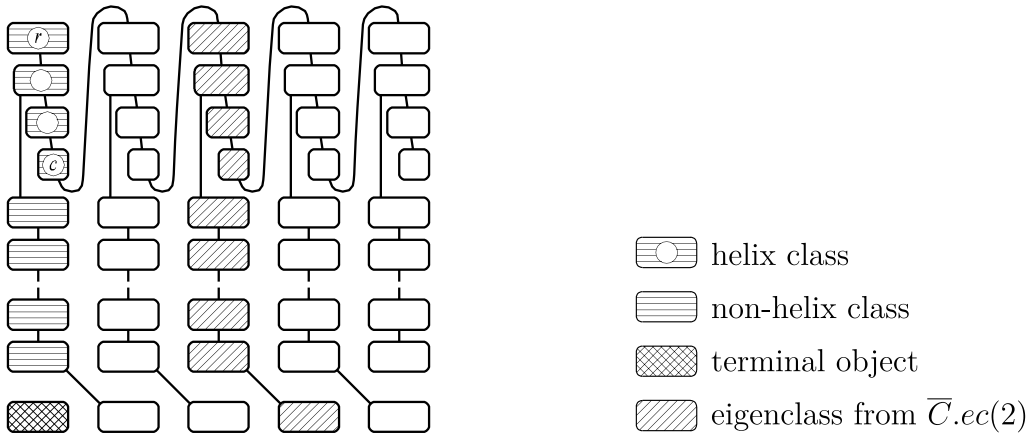

Figure 1 – Side view diagram of a generic S1 structure

3.1 The Ruby helix

The upper part of figure 1 provides an explanation for used terminology. It shows that an S1 structure contains a built-in substructure (the *Ruby helix*) that resembles a helical threading of a right-infinite screw.

Proposition Let $\mathcal{S}$ be an S1 structure.

(1) Any substructure of $\mathcal{S}$ is an S1 structure.
(2) The set $\underline{H}$ of helix objects forms the smallest substructure of $\mathcal{S}$.
(3) $(\underline{H}, .sc, \underline{r})$ is a tree that is a chain.

Proof: (3) For each $i \geq 0$, $\underline{H}.pr.ec(i)$ are chains in $.sc$, with $\underline{c}.ec(i)$ and $\underline{r}.ec(i)$ as bottom and top, respectively. By s1(3b) + s1(3c), $\underline{r}.ec(i+1).sc = \underline{c}.ec(i)$, so that the bottom of $\underline{H}.pr.ec(i)$ is the parent of the top of $\underline{H}.pr.ec(i+1)$. □

As of version 1.9, Ruby provides 4 helix classes, constituting the following chain:

$$\underline{c} = \texttt{Class} < \texttt{Module} < \texttt{Object} < \texttt{BasicObject} = \underline{r}.$$

3.2 Modules

A *module* is a terminal object x such that, for some i,

$$x.ec.sc(i) = \underline{c}.sc \ (= \texttt{Module}),$$

i.e., according to the terminology established later, a module is a terminal that is a `Module`.

Modules are distinguished terminals with additional semantics that is introduced in the S2 structure, a refinement of the S1 structure. Similarly to classes and eigenclasses, modules play the role of type system constituents, particularly in that they provide functionality (methods) for other objects. Terminals that are not modules can be regarded as “end users” of the Ruby type system and called *pure instances*.

3.3 A sample S1 structure

A sample S1 structure is shown in figure 2, this time from the "front view". The structure contains 22 primary objects: 4 helix classes, 5 other built-in classes (`String`, `Numeric`, `Integer`, `Fixnum` and `Bignum`), 2 built-in modules (numbered with 0 and 1), 4 user-created classes (`S`, `A`, `B` and `X`), 2 user-created modules (`M` and `N`), and 5 user-created terminal objects (that are not modules – `s`, `i`, `j`, `b` and `k`).

The primary objects are created by the following code:

```
class S < String; end
class A;          end
class B < A;      end
class X < Module; end
module M;         end
N = X.new
s = S.new; i = 20; j = 30; k = 2**70; b = B.new
```

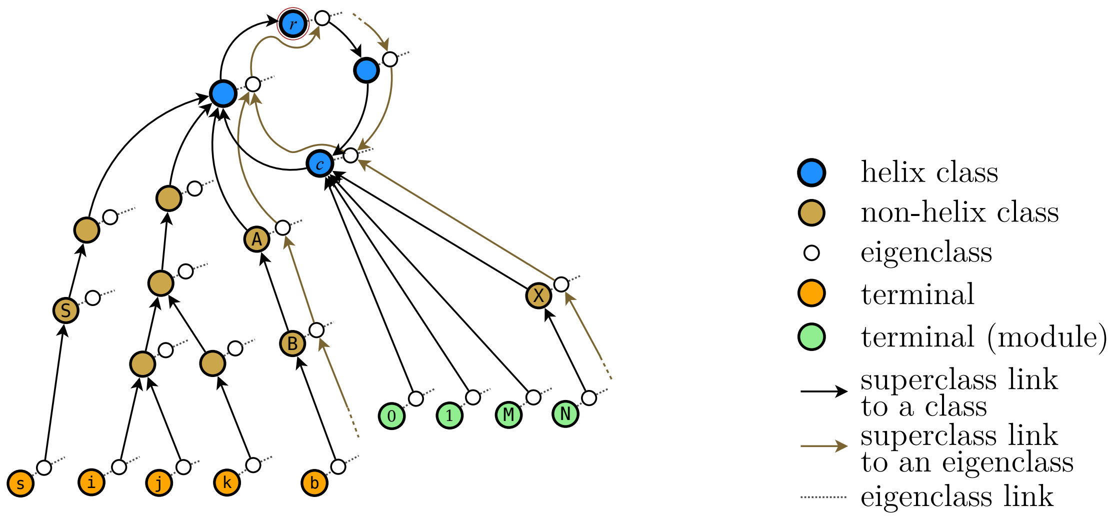

Figure 2 – Front view diagram of an S1 structure

3.4 Inheritance

Proposition $(\underline{O} \setminus \underline{T}, .sc, \underline{r})$ is an algebraic tree.

Proof: Denote $\mathcal{C}_i$ the partial algebra $(\overline{C}.ec(i), .sc)$, $i \geq 0$. By s1(3c), for each $i > 0$, $\mathcal{C}_{i-1}$ and $\mathcal{C}_i$ are isomorphic, so that they are isomorphic to $\mathcal{C}_0 = (\overline{C}, .sc)$ which is, by s1(3a), a tree with root $\underline{r}$. By s1(3b) + s1(3c), $\underline{r}.ec(i).sc = \underline{c}.ec(i-1)$, $i > 0$, so that each pair $(\underline{r}.ec(i), \underline{c}.ec(i-1))$ forms an oriented bridge (a "twist" link) from $\mathcal{C}_i$ to $\mathcal{C}_{i-1}$. This means that the partial algebra $(\underline{O} \setminus \underline{T}, .sc, \underline{r})$ is a tree.

□

Note that there are distinguished subtrees of $(\underline{O} \setminus \underline{T}, .sc, \underline{r})$, in particular those formed by sets $\underline{C}$ and $\overline{C}$.

We denote $\mathbb{H} = (\underline{O} \setminus \underline{T}, \leq)$ the reflexive transitive closure of $(\underline{O} \setminus \underline{T}, .sc)$ and call it the *superclass inheritance* or *sc-inheritance*. For every non-terminal objects x, y,

$$x \leq y \ \text{ iff } \ x.sc(i) = y \text{ for some } i \geq 0.$$

Note: In the S2 structure, the domain of $\leq$ is extended to also involve modules. This relation then corresponds to the Ruby reflection operator `<=` (method of `Module`).

Observation: Let x, y be non-terminal objects and let i, j be such that $x \in \overline{C}.ec(i)$ and $y \in \overline{C}.ec(j)$.

(i) $x \leq y$ iff $x.ec \leq y.ec$.

(ii) If $i \neq j$ then $x \leq y$ iff $i > j$ and y is a helix object. □

We denote $x.hancs$ the list corresponding to the superclass chain of a non-terminal x,

$$x.hancs[i] = x.sc(i) \ \text{ whenever } x.sc(i) \text{ is defined.}$$

The list $x.hancs$ without eigenclasses (so that it contains just classes) is denoted $x.hancestors$. This means that $x.hancs = p + x.hancestors$ for a prefix list p of eigenclasses (p is empty if x is a class).

3.5 The *.class* map

The *class* map, $.class : \underline{O} \to \underline{C}$, is defined by $x.class = x.ec.hancestors[0]$, so that the class of x is the first class (least in $\leq$) in the superclass chain of $x.ec$. Equivalently,

$$x.class = x.ec.sc \text{ if } x \text{ is terminal,}$$
$$x.class = \underline{r}.ec.sc = \underline{c} \text{ otherwise.}$$

This means that the class map forms a tree shown in figure 3.

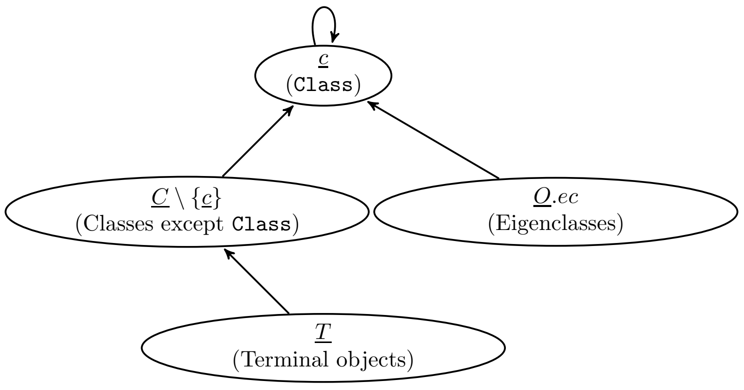

Figure 3 – The *.class* map

Observation: Up to isomorphism, an S1 structure is uniquely given by $(\underline{O}.pr, .class, .sc)$.

3.6 The instance-of and kind-of relations

The *instance-of* and (S1-) *kind-of* relations are defined as compositions $.class \circ \mathbb{H}$ and $.ec \circ \mathbb{H}$, respectively, i.e., for every object x and every non-terminal object y,

$$\begin{array}{llll} x \text{ is an instance of} & y & \text{iff} & x.class \le y, \\ x \text{ is kind of} & y & \text{iff} & x.ec \quad \le y. \end{array}$$

If $x.class = y$ then we say that x is a *direct* instance of y [2]. Instances of a class named `X` are referred to as `X`s. An instance of `X` is said to be *an* `X`.

Note: Similarly to $\le$, the kind-of relation is introduced in its restriction: in the S1 structure, it only allows non-terminal objects on its right side. The "full" kind-of also allows an object to be kind of a module. The direct-instance-of, instance-of, S1-kind-of and kind-of relations then form an inclusion chain:

$$.class \;\subseteq\; .class \circ \mathbb{H} \;\subseteq\; .ec \circ \mathbb{H} \;\subseteq\; .ec \circ (\le).$$

Observations:

(i) For every object x,

$$\begin{array}{lll} x \text{ is an instance of itself} & \text{iff} & x \text{ is a helix class}, \\ x \text{ is kind-of } ^{3} \text{ itself} & \text{iff} & x \text{ is a helix object}. \end{array}$$

(ii) Restrictions of $.class \circ \mathbb{H}$ and $.ec \circ \mathbb{H}$ to $\underline{O}.pr$ are equal.

(iii) Up to isomorphism, an S1 structure is uniquely given by $(\underline{O}.pr, .class \circ \mathbb{H}, .sc)$.

3.7 Nomenclature of objects

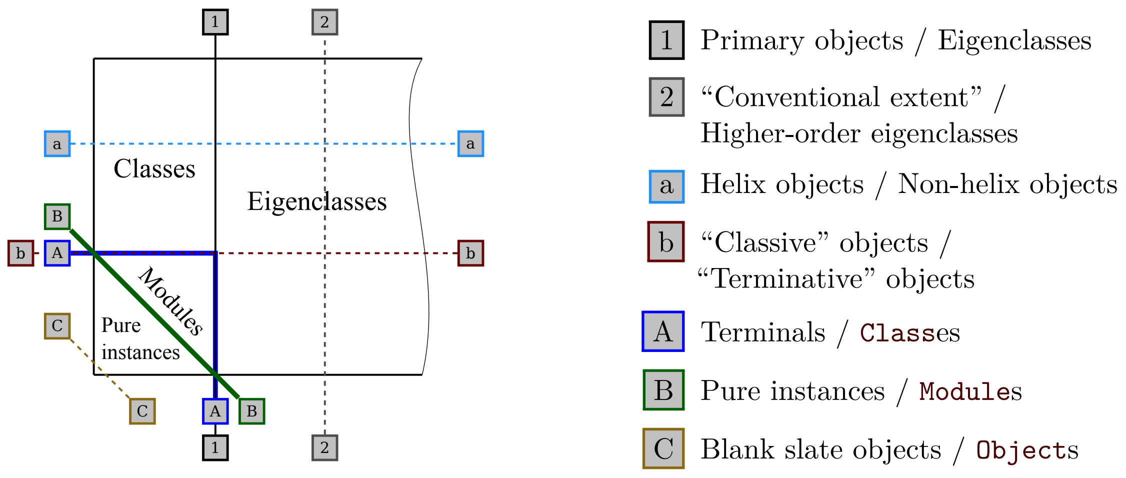
1
2
a
Classes
Eigenclasses
B
b
A
Modules
Pure instances
C
1 Primary objects / Eigenclasses
2 "Conventional extent" / Higher-order eigenclasses
a Helix objects / Non-helix objects
b "Classive" objects / "Terminative" objects
A Terminals / Classes
B Pure instances / Modules
C Blank slate objects / Objects

Figure 4 – The S1 nomenclature

Figure 4 shows a nomenclature of Ruby objects that is induced by the S1 structure. Each of the 7 division lines partitions the set of objects into two complementary subsets, specified as

[2] This corresponds to the `instance_of?` method.
[3] Again, we mean $ec \circ \mathbb{H}$ here. The "full" kind-of also allows a module to be kind of itself.

<set> / <complemetary set>. The 4 labels inside the diagram apply to regions bordered by full lines.

Note in particular that “`Classes`” is not synonymic to “Classes”, similarly for `Modules` and `Objects`.

4 S1 superstructure representation

By a *superstructure representation* of the Ruby S1 structure we mean a structure $(V, \underline{O})$ where

V is a superstructure (according to 2.7) with its set of urelements denoted U.

$\underline{O}$ is a subset of V, elements of $\underline{O}$ are called *objects*. We distinguish subsets $\underline{T}$, $\underline{C}$ and $\underline{H}$, of *$\underline{t}$erminals*, *$\underline{c}$lasses* and *$\underline{h}$elix objects*, respectively, as follows:

$\underline{T}$ consists of objects that are urelements $(\underline{T} = \underline{O} \cap U)$.
$\underline{C}$ consists of objects that contain at least 2 urelements.
$\underline{H}$ consists of objects that have at least 2 strata.

The structure is subject to the following conditions:

s1-rep(0) The set $\underline{O}.pr$ of primary objects is finite.

s1-rep(1) $\underline{O}.pr = \underline{C} \uplus \underline{T}$ and $\underline{O}.ec \subset \underline{O}$, i.e. for every object x,

the primary element $x.pr$ is a class or a terminal,
the eigenclass $x.ec$ is an object.

s1-rep(2)

The set $\underline{C} \cap \underline{H}$ of helix classes is a finite set $\{\underline{c} = h_0, h_1, \ldots, h_{n-1} = \underline{r}\}$, $n \geq 2$, such that

$$h_i = U_i \uplus \mathbb{P}_+(\mathbb{P}_\star^{\underline{k}-2}(U))$$

for some $\underline{k} \geq 2$ and some sets of urelements $U_0, U_1, \ldots, U_{n-1}$, $i = 0, \ldots, n-1$, such that

(a) $U_0 \subset U_1 \subset \cdots \subset U_{n-1} = U$ (in particular, $\underline{c} \neq \underline{r}$),
(b) U_0 is disjoint with $\underline{T}$ and with every non-helix class.

s1-rep(3)

$(\underline{C} \cup U.ec, \subseteq)$ is a forest – therefore $(C \cup U.ec, \subseteq, \underline{r})$ is an algebraic tree. [4]

We call the number $\underline{k}$ the *stratality* of $(V, \underline{O})$. We also denote $\overline{C} = \underline{C} \cup \underline{T}.ec$, a set that forms a subtree of $(C \cup U.ec, \subseteq, \underline{r})$.

Observations:

(i) $\underline{r} = \bigcup \underline{C} = \mathbb{P}_\star^{\underline{k}-1}(U)$.

(ii) $\underline{c} = \bigcap \underline{H}.pr$, $\underline{H}.pr = \underline{C} \cap \underline{H}$.

(iii) Terminals / non-helix classes / helix classes have rank 0 / 1 / $\underline{k}$, respectively.

[4] Recall that $U.ec$ is the set of all singleton subsets of U.

(iv) Non-terminals x are either mono-stratal (if $x \notin \underline{H}$) or $\underline{k}$-stratal (if $x \in \underline{H}$).

(v) For every non-terminal object x and every $i \geq 0$,

$$x \in \overline{C}.ec(i) \ \text{ iff } \ x.\rho.d = i + 1.$$

(vi) Condition $\textbf{s1-rep}(1)$ means that $(\underline{O}, .ec, .pr)$ is a subalgebra of $(V, .ec, .pr)$.

(vii) Condition $\textbf{s1-rep}(2b)$ implies that the $\underline{c}$ class has the following constraints:

- There is no class x such that $x \subset \underline{c}$.
- There is no terminal x such that $x \in \underline{c}$.

(viii) The bottom stratum operator $.\rho$, in a restriction, is an order-embedding of $(\underline{C} \cup U.ec, \subseteq)$ into $(\mathbb{P}_{+}(U), \subseteq)$. In particular, $(\overline{C}, \subseteq)$ is isomorphic to $(\overline{C}.\rho, \subseteq)$.

We define the *superclass* map $.sc : \underline{O} \setminus (\{\underline{r}\} \cup \underline{T}) \to \underline{O}$ in a correspondence to $\textbf{s1}(3)$:

$x.sc$ equals the parent of x in $(\overline{C}, \subseteq)$ if $x \in \overline{C} \setminus \{\underline{r}\}$,
$x.sc$ is defined just like in $\textbf{s1}(3bc)$ if $x \in \underline{O}.ec \setminus \underline{T}.ec$.

Proposition A $(\underline{O}, .ec, .pr, .sc, \underline{r}, \underline{c})$ is an S1 structure.

Proof: The proof is a straightforward verification of $\textbf{s1}(0)$–$\textbf{s1}(4)$. □

Proposition B Every S1 structure $\mathcal{S}$ has a superstructure representation, for any given stratality $\underline{k} \leq 2$.

Proof: Given an S1 strucure $\mathcal{S}$, with all the established notation, and a natural $\underline{k} \geq 2$, we construct its representation $(V, \underline{O}.\nu)$ as follows. Let $U = \underline{T}.ur \uplus \underline{C}.ur(1) \uplus \underline{C}.ur(2)$ be the set of urelements of a superstructure V where $.ur : \underline{T} \hookrightarrow U$ and $.ur() : \underline{C} \times \{1, 2\} \hookrightarrow U$ are injective maps (so that U contains exactly one copy of each terminal and two copies of each class).

We define an injective map $.\nu : \underline{O} \hookrightarrow V$ by

(a) If a is a class then $a.\nu$ is a subset of $\mathbb{P}_{\star}^{\underline{k}-1}(U)$ such that $x \in a.\nu$ iff one of the following conditions is satisfied:

(i) $x = y.ur(i)$ and $y \leq a$ for some $y \in \underline{C}$, $i \in \{1, 2\}$.
(ii) $x \in \underline{T}$ and $x.ec \leq a$.
(iii) $x \in \mathbb{P}_{+}(\mathbb{P}_{\star}^{\underline{k}-2}(U))$ and a is a helix class.

(b) If a is terminal then $a.\nu = a.ur$.

(c) If a is an eigenclass then $a.\nu = a.pr.\nu.ec(i)$ where $i = a.eci$.

The structure $(V, \underline{O}.\nu)$ is then a superstructure representation of $\mathcal{S}$, with $.\nu$ being an isomorphism between $\mathcal{S}$ and the S1 structure induced by $(V, \underline{O}.\nu)$. □

4.0.1 Urobject

We denote $\underline{u} = \mathbb{P}_{\star}^{\underline{k}-2}(U)$ and call this element *the urobject.* Obviously, $\underline{u}$ is a primary element that is not an object (neither it is an urelement). By $\textbf{s1-rep}(2)$,

$$h = h.\rho \uplus \underline{u}.ec \ \text{ for every helix class } h.$$

4.1 An example

Figure 5 shows an S1 superstructure representation with 15 urelements, 2 terminals (`b` and `M`) 7 classes (4 helix classes and `A`, `Q`, `B`). Except for terminals, object eigenclasses are not displayed.

The structure can be created by the following code:

```
class A;      end
class Q < A;  end
class B < A;  end
b = B.new
module M;     end
```

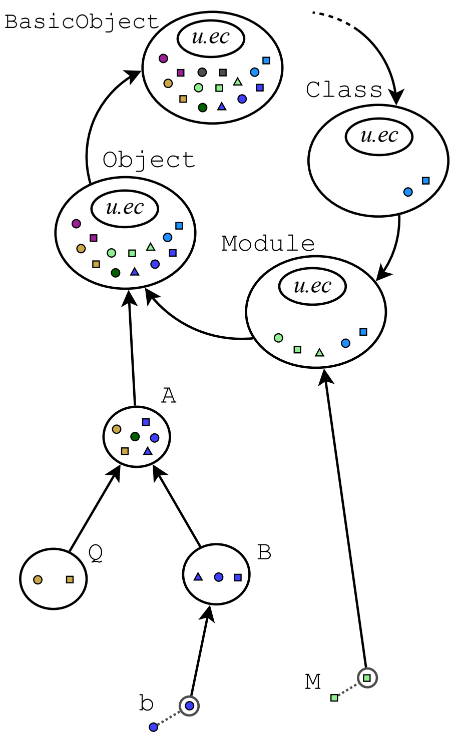

Figure 5 – S1 representation example

4.2 Semantics of ⊆ and ∈

In the following, we show how

- inheritance ($\mathbb{H}$) corresponds to set inclusion ($\subseteq$), and
- the kind-of relation ($.ec \circ \mathbb{H}$) corresponds to set membership ($\in$).

Proposition C For every objects x, y,

$$x.\overline{\rho} \in y \ \text{ iff } \ x.ec.\rho \subseteq y,$$

Proof: (a) If y is a terminal then neither side of "iff" can be satisfied.
(b) If y is an eigenclass then $x.\overline{\rho} \in y$ iff $x.\overline{\rho}.ec \subseteq y$ iff $x.ec.\rho \subseteq y$.
(c) If x is a terminal then $x.\overline{\rho} = x \in y$ iff $x.ec.\rho = \{x\} \subseteq y$.
(d) If y is a non-helix class and x a non-terminal then neither side of "iff" can be satisfied.
(e) Let y be a helix class, x a non-terminal. Then $x.\rho \in y$ iff $x.\rho \in \underline{u}.ec$ iff $x.\rho.ec \subseteq \underline{u}.ec$ iff $x.ec.\rho \subseteq \underline{u}.ec$ iff $x.ec.\rho \subseteq y$. □

Proposition D Let $\underline{O}_{<i}$ denote the set of objects with the eigenclass index less than i.

(1) For every $x, y \in \underline{O}_{<\underline{k}} \setminus \underline{T}$,

$$x \leq y \quad \text{iff} \quad x.\rho \subseteq y,$$

i.e. x is an (*.sc*-)inheritance descendant of y iff the bottom stratum of x is a subset of y.

(2) For every $x \in \underline{O}_{<\underline{k}}$, $y \in \underline{O}_{<\underline{k}} \setminus \underline{T}$,

$$x.ec \leq y \quad \text{iff} \quad x.\overline{\rho} \in y,$$

i.e. x is kind-of y iff x or its bottom stratum is an element (a member) of y.

(3) For every $x \in \underline{O}_{<\underline{k}}$,

$$x.class = \bigcap \{y \in \underline{C} \mid x.\overline{\rho} \in y\},$$

i.e. the class of x is the smallest class (w.r.t. inclusion) that contains x or its bottom stratum.

Proof:

(1) Let $x, y \in \underline{O}_{<\underline{k}} \setminus \underline{T}$ and denote $i = x.\rho.d - 1$, $j = y.\rho.d - 1$, so that $x \in \overline{C}.ec(i)$, $y \in \overline{C}.ec(j)$.

(a) Let $i = j = 0$, i.e. $x, y \in \overline{C}$. Then

$$x \leq y \ \text{ iff } \ x \subseteq y \ \text{ iff } \ x.\rho \subseteq y.$$

The first equivalence is by definition: $(\overline{C}, \leq)$ is the reflexive transitive closure of $(\overline{C}, .sc)$ which is defined as the reflexive transitive reduction of $(\overline{C}, \subseteq)$. The second equivalence is immediate if x is not a helix class, that is, if $x.\rho = x$. If x is a helix class then $x.\rho \subseteq y$ implies that y is also a helix class, since, by **s1-rep**(2b), non-helix classes are disjoint with $\underline{c}.\rho$, a subset of $x.\rho$. It follows that $x \subseteq y$.

(b) Let $i = j > 0$, i.e. $x = a.ec(i)$, $y = b.ec(i)$ for some $a, b \in \overline{C}$. Then $a.ec(i) \leq b.ec(i)$ iff $a \leq b$ iff $a \subseteq b$ iff $a.\rho \subseteq b$ iff $a.\rho.ec(i) \subseteq b.ec(i)$ iff $a.ec(i).\rho \subseteq b.ec(i)$.

(c) Let $i \neq j$. By observation (ii) in 3.4,

$$x \leq y \ \text{ iff } \ i > j \text{ and } y \text{ is a helix object.}$$

We show that the same holds with "$x \leq y$" replaced by "$x.\rho \subseteq y$".

(c1) Let $i < j$, equivalently, $x.\rho.d < y.\rho.d$. This implies $x.\rho \not\subseteq y$.

(c2) Let $i > j$ and $y \notin \underline{H}$. Then $x.\rho.d > y.\rho.d = y.d$, thus $x.\rho \not\subseteq y$.

(c3) Let $i > j$ and $y \in \underline{H}$. Then $x = a.ec(j+1)$ for some $a \in \overline{C}.ec(i - j - 1)$, $y = b.ec(j)$ where $b = y.pr = b.\rho \uplus \underline{u}.ec$ is a helix class. Since $\underline{u}.d = \underline{k} - 1$ and $a.\rho.d < x.\rho.d \leq \underline{k}$, it follows by the definition of the urobject $\underline{u}$ that $a.\rho \subseteq \underline{u}$. We then obtain

$$x.\rho = a.ec(j+1).\rho = a.\rho.ec(j+1) \ \subseteq \ \underline{u}.ec.ec(j) \ \subseteq \ b.ec(j) = y.$$

This shows that $x.\rho \subseteq y$.

(2) Follows from (1) and proposition C.

(3) Denote $S = \{y \in \underline{C} \mid x.ec \leq y\}$. By definition, $x.class$ equals the least element of S, w.r.t. $\leq$. By (2), $S = \{y \in \underline{C} \mid x.\overline{\rho} \in y\}$. Since $\leq$ coincides with $\subseteq$ on $\underline{C}$, a superset of S, the least element of S equals $\bigcap S$.

$\square$

5 Object actuality

Because eigenclass chains are infinite, any implementation of the S1 structure must involve lazy creation – for a primary object x, only finitely many eigenclasses are actually created. This “actuality” state can be regarded as a refinement of the S1 structure.

We express this refinement by a set $\underline{O}_{\mathrm{a}}$, called the *actuality extent*. Elements of $\underline{O}_{\mathrm{a}}$ are *actual(s)*. The following conditions are required:

s4(1) $\underline{O}_{\mathrm{a}}$ is finite.
s4(2) $\underline{O}.pr \subseteq \underline{O}_{\mathrm{a}}$. (Primary objects are actual.)
s4(3) $\underline{O}_{\mathrm{a}}.ce \subset \underline{O}_{\mathrm{a}}$. (The actual part of an eigenclass chain is its starting part.)
s4(4) $\underline{O}_{\mathrm{a}}.sc \subset \underline{O}_{\mathrm{a}}$. (Actual objects form a subtree of superclass inheritance.)
s4(5) $\underline{r}.ec(i) \in \underline{O}_{\mathrm{a}}$ implies $\underline{c}.ec(i) \in \underline{O}_{\mathrm{a}}$. (Helix actual lists are equally sized.)
s4(6) $\underline{c}.ec \in \underline{O}_{\mathrm{a}}$. (First helix eigenclasses are actual.)

Note: The use of the s4 prefix is conformant to [Pav12a] where object actuality is introduced in the S4 structure.

We say that object actuality is *conventional* if the actuality extent is within the conventional extent, i.e. $\underline{O}_{\mathrm{a}} \subseteq \underline{O}_{01}$ – higher order eigenclasses are not actual. This condition is satisfied in most Ruby programs.

For a primary object x, we denote $x.actuals$ the list corresponding to the finite eigenclass subchain of actual objects starting at x. Conventional actuality is then equivalent to the equality $\underline{c}.actuals.last = \underline{c}.ec$.

5.1 The actualclass map

For an object x we define $x.aclass$, the *actualclass* of x, to be the first member of $x.ec.hancs$ that is actual. Semantically, object’s actualclass is the (conceptual) actual startpoint of method lookup – non-actual eigenclasses before $x.aclass$ can be skipped.

Figure 6 shows the position of $x.aclass$ in $x.ec.hancs$ for $x =$ `B` in a structure created by `class A; end; class B < A; end` (see also figure 7(a)).

Observations:

(i) For every object x, $x.ec \leq x.aclass \leq x.class$.

(ii) The actualclass map can be recursively defined by

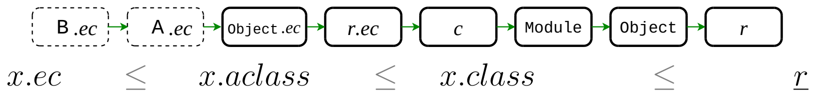

Figure 6 – The position of $x.aclass$ in the superclass chain of $x.ec$

$x.aclass = x.ec$ if $x.ec$ is actual, else
$x.aclass = x.ec.sc$ $(= x.class)$ if x is terminal, else
$x.aclass = x.sc.aclass$.

(iii) $(\underline{O}, .aclass)$ is a tree such that

the root equals $\underline{c}.actuals.last$ ($\underline{c}.ec$ under conventional actuality),
terminals have depth $\underline{c}.actuals.length + 1$ (3 under conventional actuality).

5.2 An example

Figure 7 shows several cases of object actuality in a sample S1 structure. Arrows indicate the actualclass map along the chain starting in `b`. Except for the root loop at $\underline{c}.ec$, the arrows are drawn in a way to intersect skipped eigenclasses. Cases (a)–(d) can be incrementaly created by the following code.

```
class A;      end
class B < A; end
b = B.new              # (a) O_a = O.pr ∪ H.pr.ec
class << A; end        # (b) ··· ∪ {A.ec}
class << B; end        # (c) ··· ∪ {B.ec}
class << b; end        # (d) ··· ∪ {b.ec}
```

The (e) case is created by skipping the line for the (c) case.

5.3 The $.klass$ map

The transition from (a) to (b) displayed in figure 7 shows that there is an efficiency problem with implementing the $.aclass$ map directly as a pointer structure. Updating the value of `B`$.aclass$ (from `Object`$.ec$ to `A`$.ec$) would require a traversal of all descendants of `A`. This is solved in MRI 1.9 by allocating additional eigenclasses, which can be considered *semiactual*. (They are displayed with a dashed border in figure 7.) Each class has exactly one semiactual eigenclass in its eigenclass chain, so that the set of semiactual objects can be expressed as $\underline{C}.actuals.last.ec$.
Note: We assume that the set $X = \underline{T}.ec(2) \cap \underline{O}_{\mathrm{a}}$ is empty, i.e. second eigenclasses of terminals are not actual. In general, the set of semiactual objects also includes $X.pr.actuals.last.ec$. Moreover, an additional constraint is imposed by MRI to object actuality which only manifests itself if X is non-empty.

Let $\underline{O}_{\mathrm{a}*}$ be the set of objects that are actual or semiactual. The actualclass map is then implemented via the $.klass$ map which is a map $\underline{O}_{\mathrm{a}} \to \underline{O}_{\mathrm{a}*}$ such that

$x.klass = x.ec$ if $x.ec$ is semiactual,

$x.klass = x.aclass$ otherwise.

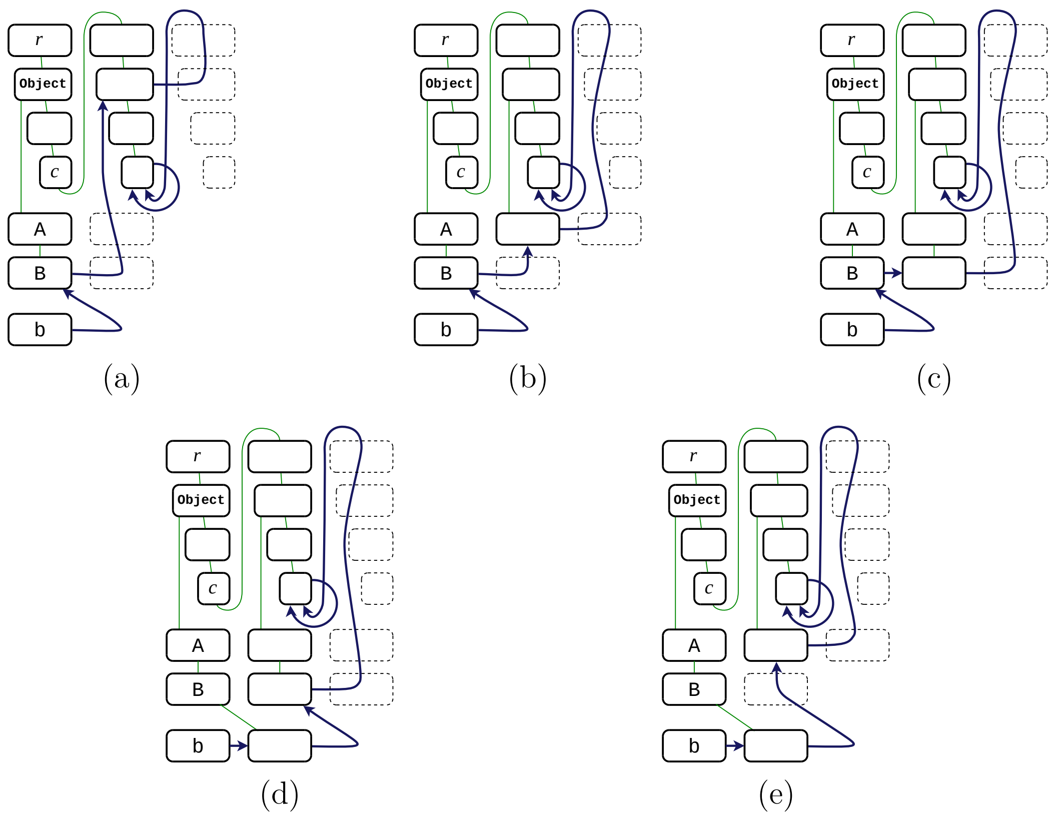

Figure 7 – Eigenclass actualization

6 Correspondence with Smalltalk-80

The Ruby S1 structure can be seen as a rectification of the correspondent structure from the Smalltalk-80 object model. For a detailed comparison, see [Pav12b].

In Smalltalk, the following requirements are realized:

(0) There are 3 basic types of objects:

- (a) *Terminal objects.* These are objects that are not (direct) method providers.
- (b) *Classes.* These are non-terminal objects with built-in naming support.
- (c) Non-terminal objects without naming support. In Smalltalk, such objects are called *metaclasses.*

Note: As a rule, Smalltalk literature [GR83] [Hun97] is rather dialectical in terminology for (b) and (c).

(1) There is a kind-of relation between objects. For objects x, y, x is kind-of y means that y provides methods for x as the receiver.

(2) There is an inheritance between non-terminal objects. The inheritance, denoted $\mathbb{H}$, is a partial order that is a tree.

(3) For every object x, the set $\{y \mid x \text{ is kind-of } y\}$, denoted $\mathrm{mlc}(x)$, is an up-set chain in $\mathbb{H}$, the *method lookup chain* for x as the receiver.

(4) For every classes x, y, if $(x, y) \in \mathbb{H}$ then $\mathrm{mlc}(x) \supseteq \mathrm{mlc}(y)$.

(5) Different classes have different method lookup chain.

We denote $x.aclass$ the least (i.e. first) element of $\mathrm{mlc}(x)$. In Smalltalk, this map is reflected by a method named `class`. Conditions (0)–(5) are achieved by the structure schematized in figure 8. For each class x, $x.aclass$ is the unique metaclass for x. Metaclasses, on the other hand, have constant method lookup chain: for each metaclass x, $x.aclass$ equals the `Metaclass` class. For terminal objects x, $x.aclass$ is a class.

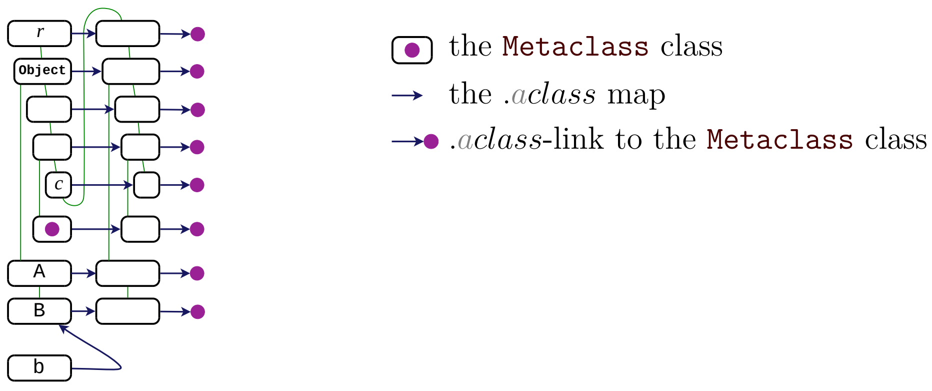

Figure 8 – Smalltalk-80 analogue of the S1 structure

In the Ruby object model, (4) and (5) are replaced by stronger conditions:

(4*) For every $\underline{\text{objects}}$ x, y, if $(x, y) \in \mathbb{H}$ then $\mathrm{mlc}(x) \supseteq \mathrm{mlc}(y)$. [5]

(5*) Different $\underline{\text{objects}}$ have different method lookup chain.

This is achieved by introducing *eigenclasses* as unique meta-objects. There is nothing special with classes regarding their ownership of meta-objects. Also terminal objects have meta-objects as well as the meta-objects themselves.

This can be viewed as a rectification of conceptual inconsistencies of the Smalltalk-80 object model:

- The Ruby `class` method works according to its name – applied to an object x, it returns $x.class$, i.e. the first class encountered in the method lookup chain for x.

  In Smalltalk, the `class` method means in fact the (Smalltalk version of the) actualclass map, revealing implementation detail rather than providing conceptual information about the class system.

- There is no `Metaclass` class in Ruby. For eigenclasses of classes, the method lookup chain merely starts with some non-actual eigenclasses until it reaches $\underline{c}.ec$, the eigenclass of the `Class` class.

7 Transitions

We partially describe the state transition system of S1 structures by specifying the fundamental constraint for the state domain.

S1-T(1) Let $\mathcal{S}_i = (\underline{O}_i, .ec_i, .pr_i, .sc_i, \underline{r}_i, \underline{c}_i)$, $i = \{1, 2\}$, be a pair of S1 structures from the state domain. Then

[5] In Smalltalk, this condition is not satisfied if x is a helix class and y a metaclass.

$$(\underline{O}_1 \cap \underline{O}_2, .ec_i, .pr_i, .sc_i, \underline{r}_i, \underline{c}_i) \text{ is a substructure of } \mathcal{S}_i,\ i = \{1, 2\}.$$

This means that superclass and eigenclass links between objects are fixed – they are preserved by transitions. Subsequently, transitions preserve the derived maps, relations and substructures, in particular, the Ruby helix, object nomenclature, the class map and the instance-of relation.

8 Further extensions

S1 structures together with their transition system form the first non-trivial approximation of the Ruby object model as described in [Pav12a]. As the next refinement, the already mentioned S2 structure is presented. This structure allows to express inclusion of modules into `Modules` [6]. As a result, a forest structure (more precisely, a single main tree together with an isolated inclusion chain for each module) is induced, constituting a refinement of the superclass inheritance. This induced structure is the "resolution order" used for method and constant lookup.

The document [Pav12a] provides further refinements, yielding a detailed description. The resulting structure can be viewed as a naming multidigraph, consisting of nodes and uniquely labelled arrows between them. Objects are special kind of nodes, distinguished arrow names have special semantics. Incremental specification corresponds with a stratification of arrows.

9 Summary and conclusion

This document provides a rigorous description of a structure that is of fundamental importance to object-oriented programming. In its rudimentary form, the structure first appeared in the Smalltalk-80 programming language three decades ago. The rectified version has been used to build the foundation of the data model of the Ruby programming language.

As a result of the description, important maps and relations between objects have been derived, in particular, superclass inheritance, the class map, the instance-of relation and, partially, the kind-of relation. We also have established a consistent terminology for basic constituents of the Ruby object model.

In section 4, we provided a set-theoretic representation of the S1 structure. It is shown how the structure can be interpreted via the fundamental relations $\subseteq$ and $\in$ of set theory.

In section 5, we slightly refined the S1 structure by distinguishing object actuality extent to reflect the necessity of lazy creation of eigenclasses. This induces the actualclass map which is, with respect to inheritance, "positioned" between the eigenclass and the class maps. The *.klass* map is then a slight alteration of the actualclass map that closely relates to the implementation.

Section 6 reveals the correspondence of the S1 structure with its precursor in Smalltalk-80. It is shown that the Smalltalk model contains conceptual inconsistencies that have been rectified by the Ruby object model.

Hopefully, this article demonstrates at least the following two things:

- The potential of mathematical structures for a human-centric description of software.
- The exquisite quality of the Ruby programming language.

[6] For convenience, `Modules` (i.e. the set of classes, eigenclasses and modules) are called *includers*.

References

[Bla09] David A. Black. *The Well-Grounded Rubyist.* Manning Publications, 2009.

[BS03] Egon Börger and Robert Stärk *Abstract State Machines: A Method for High-Level System Design and Analysis.* Springer, 2003.

[CK90] C. C. Chang and H. Jerome Keisler. *Model Theory.* Studies in Logic and the Foundations of Mathematics. Elsevier, third edition, 1990.

[DP02] B. Davey and H. Priestley. *Introduction to Lattices and Order.* Cambridge University Press, second edition, 2002.

[FM08] David Flanagan and Yukihiro Matsumoto. *The Ruby Programming Language.* O'Reilly, 2008.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation.* Addison Wesley, 1983.

[Gur95] Yuri Gurevich. *Evolving Algebras 1993: Lipari Guide.* Specification and Validation Methods, E. Börger (ed.) Oxford University, Press 1995.

[Hun97] John Hunt. *Smalltalk and Object Orientation: An Introduction.* Springer Verlag, 1997.

[Mul10] F. A. Muller *The Characterisation of Structure: Definition versus Axiomatisation.* The Present Situation in the Philosophy of Science, F. Stadler et al. (eds.) Dordrecht: Springer Verlag, 2010.

[Pav12a] Ondřej Pavlata. *The Ruby Object Model: Data Structure in Detail.* `http://www.atalon.cz/rb-om/ruby-object-model/`.

[Pav12b] Ondřej Pavlata. *The Ruby Object Model: Comparison with Smalltalk-80.* `http://www.atalon.cz/rb-om/ruby-object-model/co-smalltalk/`.

[SSB01] Robert Stärk, Joachim Schmid, Egon Börger. *Java and the Java Virtual Machine. Definition, Verification, Validation.* Springer Verlag, 2001.